IBM VisualAge® for Java™, Version 3.5

IMS Connector for Java User's Guide and Reference

Note!

Before using this information and the product it supports, be sure to read the general information under **Notices**.

Edition notice

This edition applies to Version 3.5 of IBM VisualAge for Java and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright International Business Machines Corporation 1998, 2000. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Chapter 1. Understanding IMS Connector for Java 1
Prerequisites for using IMS Connector for Java 3
IMS Connector for Java concepts and terms 4

Chapter 2. Preparing to use IMS Connector for Java 17
Preparing your VisualAge for Java Environment 17
Preparing your WebSphere Studio Environment 21
Preparing your WebSphere Application Server Environment 22

Chapter 3. Building Java applications and servlets 31
Building Java Applications and Servlets for Conversational Transactions 32

Chapter 4. Building a Java Application to Run an IMS Transaction 35

Chapter 5. Building a Java Application to Run a Navigator 53

Chapter 6. Building a Java Application for an IMS Transaction with Multi-Segment Output Messages 69

Chapter 7. Building a Java Application for an IMS Transaction with Multi-Segment Input Messages 75

Chapter 8. Building an Application to Run an IMS Transaction with Synchronization Level Confirm 81

Chapter 9. Building the Graphical User Interface 87

Chapter 10. Building a Java Servlet to Run an IMS Transaction 95

Chapter 11. Building a Web Application that Uses One Servlet to Run an IMS Conversation 107

Chapter 12. Building a Web Application that Uses Two Servlets to Run an IMS Conversation 121

Appendix A. IMS INSTALL/IVP Sample Application 129

Appendix B. IMS Connector for Java Conversational Programming Models 131

Appendix C. Using the trace and error logging facility 137
Customizing the trace facility 137
Reading the trace log 138
Reading the error log 139

Appendix D. Diagnosing problems related to sockets 141
Diagnosing problems related to connection pooling 142

Appendix E. Messages and exceptions 143

Notices 153
Programming interface information 154
Trademarks and service marks 155

© Copyright IBM Corp. 1998, 2000

Chapter 1. Understanding IMS Connector for Java

The purpose of this documentation is to help you create Java application programs or servlets that access IMS transactions by using IMS Connector for Java with IBM VisualAge for Java, IBM WebSphere Studio, and IBM WebSphere Application Server.

IMS Connector for Java provides a way to create Java applications that can access IMS transactions. In conjunction with the VisualAge for Java development environment, IMS Connector for Java lets you rapidly develop Java applications that run your transactions. With additional support from the IBM WebSphere Studio and IBM WebSphere Application Server, you can build and run Java servlets that access your transactions from Web sites.

IMS Connector for Java provides a Common Connector Framework-compliant Java interface to IMS Connect. IMS Connector for Java is a class library that consists of two packages: **com.ibm.connector.imstoc** and **com.ibm.imstoc**. All of the classes in the com.ibm.imstoc package and many of the classes in the com.ibm.connector.imstoc package are ″support″ classes that are not used by application developers during the development of applications that use IMS Connector for Java. The classes and methods that are needed during the development process are documented with Javadoc in the Reference section of the VisualAge for Java web help. IMS Connector for Java includes the following documented classes:

- IMSCommunication
- IMSConnectionSpec
- IMSConvContext
- IMSConvHttpSessionCleanup
- IMSInteractionSpec
- IMSLogonInfoItems
- DFSMsg

To view the Javadoc for these classes, click:
Help —>Reference —> IBM APIs —> Connectors —> IMS Connector

EAB commands that use IMS Connector for Java are developed with the VisualAge for Java Command Editor. An EAB command is a composite Java bean that consists of the following:

- A Java bean (page 5) representing the input to the IMS transaction
- One or more Java beans representing the output from the IMS transaction
- An IMSConnectionSpec bean, representing the connection between the EAB command and the IMS Connect host component
- An IMSInteractionSpec bean, containing information about the type of interaction that the EAB command has with IMS via IMS Connect, as well as the name of the target IMS datastore.

In addition, the EAB command should contain a bean to process potential error or status messages from IMS. These messages begin with the three characters `DFS`; the bean that processes these messages is the DFSMsg bean. All beans that are provided by IMS Connector for Java are nonvisual beans.

Figure 1 shows an overview of three environments involved in developing Java application programs and servlets that use IMS Connector for Java to access IMS

© Copyright IBM Corp. 1998, 2000

transactions, as well as some of the class libraries that are used in each environment. A Java application typically includes a graphical user interface (GUI). A Java servlet typically uses a Web browser with HTML files as the user interface.

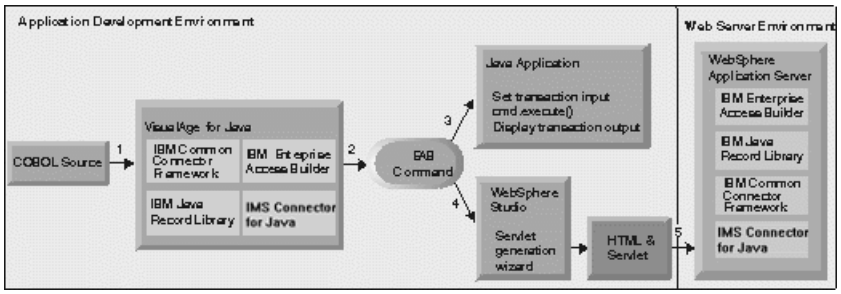


Figure 1. Developing Java Applications and Servlets Using IMS Connector for Java

The steps illustrated in Figure 1 are as follows:

1. Provide your COBOL source file to VisualAge for Java. The COBOL file contains definitions of the IMS transaction input and output messages, and might be the IMS application program that corresponds to the transaction.
2. Use the VisualAge for Java Command Editor to create an Enterprise Access Builder (EAB) command. The EAB command can be used to build either a stand-alone Java application program or a Java servlet that accesses IMS transactions.
3. VisualAge for Java can be used to quickly build and run a Java application from the EAB command.
4. If you are building a servlet, provide the EAB command to WebSphere Studio and its servlet generation wizard to generate HTML and .jsp files and a Java servlet.
5. Move the HTML and .jsp files and the Java servlet to the WebSphere Application Server environment to enable users to access your IMS transactions from a Web browser.

Figures 2 and Figure 3 show how you run either a stand-alone Java application program or a Java servlet to access your IMS transactions.

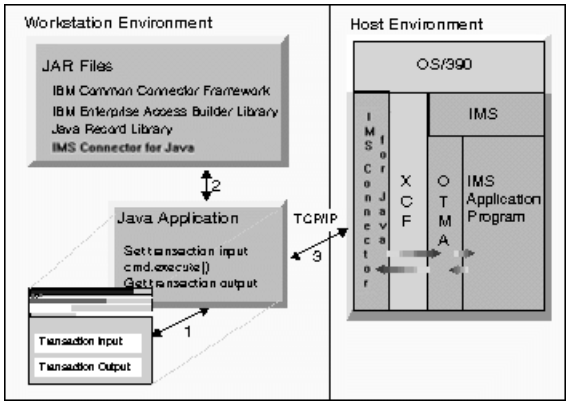


Figure 2. Running a Java Application That Uses IMS Connector for Java

In Figure 2, the steps involved are explained as follows:

1. A GUI is used to provide the IMS transaction input and to display the output from the transaction.

2. If the environment of the workstation that is running the Java application does not include all of the class libraries used by the Java application, these class libraries can be deployed onto the workstation in the form of .jar files.
3. The execute method of the EAB command uses IMS Connector for Java to submit the IMS OTMA message that runs the IMS transaction.

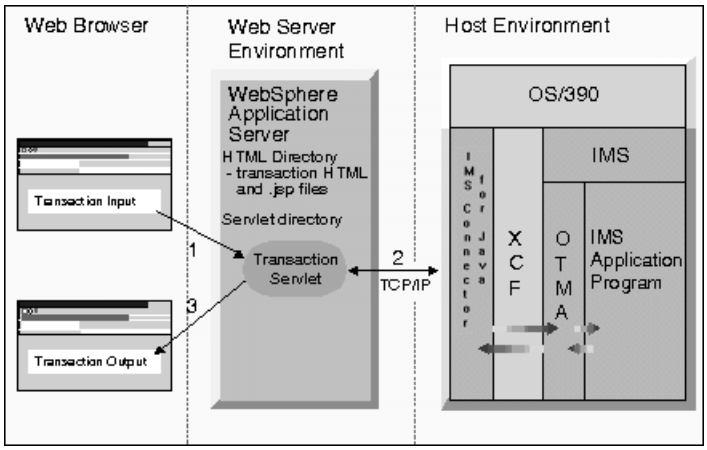


Figure 3. Running a Servlet That Uses IMS Connector for Java

In Figure 3, the steps involved are explained as follows:

1. The input to the IMS transaction is provided on the Web browser in the input HTML file. WebSphere Application Server schedules the servlet.
2. The execute method of the EAB command in the servlet uses IMS Connector for Java to submit the IMS OTMA message that runs the IMS transaction.
3. The output of the IMS transaction is displayed on the Web browser.

Prerequisites for using IMS Connector for Java

You can use IMS Connector for Java in the following three environments:

- The VisualAge for Java environment enables you to develop Enterprise Access Builder (EAB) commands that use IMS Connector for Java to access IMS transactions. VisualAge for Java also allows you to run programs that you develop from these commands.
- The WebSphere Studio environment enables you to develop Java servlets from the EAB commands that you develop in VisualAge for Java.
- The WebSphere Application Server enables you to run servlets that you develop in WebSphere Studio.

A prerequisite to using IMS Connector for Java is IMS Connect (formerly called IMS TCP/IP OTMA Connection, or simply ITOC). IMS Connect allows client applications to send messages to IMS TM through the IMS Open Transaction Manager Access (OTMA) interface, providing connection to IMS transactions from a variety of platforms, including both workstation and mainframe products. Beginning with the release of IMS Version 7, ITOC will be called IMS Connect, and will be offered as a separately priced feature of IMS. IMS Connect will provide enhancements in usability, performance, and SMP installability.

Before you attempt to run a Java application program or servlet that uses IMS Connector for Java, Version 3.5, be
sure that the following products are installed on the target host machine:

- IMS Version 7 Connect Feature and IMS Version 7 (recommended); or
- IMS Version 7 Connect Feature and IMS Version 5.1 or 6.; or
- IMS TCP/IP OTMA Connection (IMS TOC) Version 2.1.3 or above and IMS Version 5.1 or above. (IMS TOC will continue to be supported for a limited time after the general availability of IMS Version 7. See the IMS Web pages at http://www.ibm.com/software/data/ims for further details.)

IMS Connector for Java concepts and terms

This section provides an overview of some of the concepts and terminology needed to understand IMS Connector for Java, and includes:

- MFS Formatting
- Java Classes that are Provided with IMS Connector for Java
- IMS Messages
- IMS Messages Format
- IMS Logon Information
- Synchronization Level
- Connection Management
- IMS Conversations

An EAB command for a typical IMS transaction includes Java beans that represent the IMS transaction input and output messages. These beans are constructed by providing the COBOL source of the IMS transaction's application program to the VisualAge for Java COBOL parser.

The degree to which VisualAge for Java can create Java beans that accurately represent an IMS transaction's input and output messages is limited by the way in which the COBOL application program defines the messages. To overcome these limitations if any, you can create a small skeleton COBOL program that includes only the 01 level definitions for the input and output messages used by your production COBOL application program. You can then refine these definitions as necessary and use this program as input to the VisualAge for Java COBOL parser. An example of this technique can be found in Step 3: Create a COBOL RecordType Class for the IMS Transaction Input Message (page 37) under Building a Java Application to Run an IMS Transaction.

Related Reading: For more information on the COBOL parser and its limitations, see the VisualAge for Java documentation, under *VisualAge for Java Concepts, Access To the Enterprise, Enterprise Access Builder*.

MFS Formatting

Transaction input and output messages that are provided to IMS through OTMA bypass online MFS processing. MFS is the online processing component in IMS that performs message formatting, such as field padding, truncation, justification, and insertion of literal data in messages.

Currently, although VisualAge for Java includes an MFS parser, it does not support creating input and output record beans that are based on MFS message definitions (the MIDs and MODs in an MFS source file), and no MFS-like formatting is performed.

When using IMS Connector for Java you need to consider any dependencies the IMS application program has on MFS online formatting and decide whether or not you want to perform this function in your Java servlet or application.

Java Classes that are Provided by IMS Connector for Java

IMS Connector for Java provides a number of Java beans to aid you in building Java programs and servlets. All of these beans are in the IMS Connector for Java package **com.ibm.connector.imstoc**. You can combine these beans into a composite bean that accesses an IMS transaction. The beans are available to VisualAge for Java's Command Editor and Visual Composition Editor.

Definitions:

A *Java bean* is a reusable software component, an element of a program that may or may not be seen by the end user at run time and is made up of one or more Java classes. A visual Java bean is an element of a program that is seen by the end user at run time while a nonvisual bean is an element of a program that is not necessarily seen by the end user.

1. *Visual Java beans* can be used to construct a graphical user interface for an EAB command. 32 The interface can provide input to and display output from an IMS transaction. An example of a visual Java bean is a button that the user clicks to perform some action.
2. *Nonvisual Java beans* are beans which do not have visual representations at runtime. For example, all beans that are provided by IMS Connector for Java such as IMSConnectionSpec and IMSInteractionSpec, are nonvisual beans.
3. A *composite Java bean* is a Java bean that is comprised of other Java beans, either visual or nonvisual. In the context of this document, an Enterprise Access Builder (EAB) command is a composite bean that is constructed using the Visual Age for Java Command Editor.

The following classes are supplied by IMS Connector for Java:

IMSConnectionSpec

The IMSConnectionSpec bean provides information about the connection between a Java program and an IMS Connect host component, as well as information about connection management. The IMSConnectionSpec **Host name** and **Port** properties are specific to IMS Connector for Java, while the other properties are inherited from the Common Connector Framework interface **com.ibm.connector.ConnectionSpecManagementProperties**. IMSConnectionSpec properties include the following:

Host name
The TCP/IP host name of the machine running the IMS Connect that the EAB command (IMS transaction) will be using.

Port The port on the host machine where IMS Connect is running which the EAB command will use for communication with IMS Connect.

IMSConvContext

The IMSConvContext bean is used to comply with the IMS Connect requirement that the same connection is used for all iterations of an IMS conversation. A connection is a communications link, a socket in the case of TCP/IP, and is analogous to the phone line that connects two telephones during a telephone conversation. IMS Connector for Java includes this class

in its programming model for use by conversational Web applications. A Java application or servlet should create a single instance of the IMSConvContext class at the start of a conversation and associate this single instance with the connection used for the conversation. This will ensure that the connection will be preserved for the lifetime of the IMS conversation and that the CCF ConnectionManager will always return the same connection for each iteration of the IMS Conversation.

IMSConvHttpSessionCleanup
IMS Connector for Java includes this class in its programming model for use by conversational Web applications only. This class implements the HttpSessionBindingListener interface and is used to capture the unbound event of an HttpSession object during an active conversation, and to then perform appropriate cleanup.

WebSphere Application Server creates an HttpSession object to represent the HTTP session of the Web application (the connection between the browser and WebSphere Application Server.) An HttpSession object fires an unbound event when the associated HTTP session becomes unbound. When an HTTP session becomes unbound the connection needs to be returned for reuse and the IMS conversation needs to be terminated. This cleanup work is performed by methods of the IMSConvHttpSessionCleanup class.

For example, an HTTP session becomes unbound when the user exits the browser while the IMS conversation is still active. When the user prematurely exits the browser, the HTTP session associated with the browser eventually times out, as there is no more user interaction from the browser. The HttpSession object associated with the HTTP session becomes unbound. The IMSConvHttpSessionCleanup object receives an unbound event, which in turn triggers the calling of its valueUnbound() method. The valueUnbound() method issues a MODE_END_CONVERSATION request to end the IMS conversation and also performs any cleanup required to allow the connection to be reused.

IMSConvHttpSessionCleanup implements the HttpSessionListener interface and has the following methods:

void setConvContext()
Sets the IMSConvContext object of the current IMS conversation in the IMSConvHttpSessionCleanup object such that the appropriate resources can be cleaned up following an unbound event.

void valueUnbound()
This method implements the valueUnbound() method of the HttpSessionListener interface to terminate the IMS conversation and cleanup conversation resources.

IMSInteractionSpec
The IMSInteractionSpec bean provides information about the interaction between a Java program and a datastore. Interaction properties include:

ConvTerminated
This property is set to TRUE if the host IMS application program ends the IMS conversation. The Java application program checks the value of this property after invocation of the execute() method for the EAB command, to determine whether or not the conversation has been terminated by the host. The following two methods can be used:

void setConvTerminated(boolean)
Sets the value of the convTerminated property. This property is set to TRUE when the conversation is terminated by the host.

boolean getConvTerminated()
Gets the value of the convTerminated property. It is used by the Java application program to determine whether or not the conversation has been terminated by the host.

Datastore name
The name of the target IMS datastore that is defined in the IMS Connect configuration file.

LTERM Name
The LTERM name used to override the LTERM name in the IMS application program I/O PCB. The override is used if the client does not want to override the LTERM name in the I/O PCB with the transaction pipe.

Map name
Also known as the MFS MOD name, the map name can be provided by an IMS application program when returning the output of a transaction. It can also be provided by IMS when returning a status message, such as the output from a **/DIS** command, or an error message.

Mode The type of interaction to be carried out between the Java program and the IMS datastore. The modes that are currently supported include `MODE_SEND_RECEIVE`, `MODE_ACK`, `MODE_NACK`, and `MODE_END_CONVERSATION`.

Synchronization Level
Specifies the transaction synchronization level—the way in which the client (a Java application or servlet) and server transaction program (for example, an IMS application program) interact with respect to transaction output messages. Stated simply, the synchronization level determines whether or not transaction output messages must be acknowledged, that is accepted (ACK) or rejected (NACK) by the client.

DFSMsg
The DFSMsg bean represents IMS status or error messages that are returned to a Java application or servlet in response to a command or a transaction. These messages typically begin with the characters `DFS`. Often times, DFS messages are returned to a Java application or servlet. Some DFS messages indicate error situations, while others are returned as the output of IMS commands. In all cases, because OTMA is used to return the message, MFS formatting is not performed. However, IMS includes an MFS MOD name (map name) with DFS messages. IMS Connector for Java checks the MOD name field of messages that it receives from IMS Connect. If the MOD has one of the following names, the message is processed as a DFS message. If the name is not listed below, it is processed as transaction output.

- DFSMO1
- DFSMO2

- DFSMO3
- DFSMO4
- DFSMO5
- DFSDSPO1

IMS Connector for Java builds a buffer containing one or more segments of a DFS message, up to a maximum of 22 segments. Any output data beyond the 22nd segment is discarded. The buffer structure is as follows:

DFSLL1
Two-byte length (LL) of the first segment of the output message. The length (LL) of a message segment includes the length of the LL and ZZ fields of the segment.

DFSZZ1
Two-byte flag field (ZZ) of the first segment of the output message.

DFSDATA1
The data of the first segment of the output message, blank-padded or truncated to 119 bytes. `DFSLL1` represents the length of the segment received from IMS or the truncated length and, as such, is less than or equal to 123 (119 + 4).

...

...

...

DFSLL22
Two-byte length (LL) of the 22nd segment of the output message. The length (LL) of a message segment includes the length of the LL and ZZ fields of the segment.

DFSZZ22
Two-byte flag field (ZZ) of the 22nd segment of the output message.

DFSDATA22
The data of the 22nd segment of the output message, blank-padded or truncated to 119 bytes. `DFSLL22` represents the length of the segment received from IMS or the truncated length and, as such, is less than or equal to 123 (119 + 4).

The buffer structure is used to populate a DFSMsg object. A Java program can obtain a DFS message from a DFSMsg object in the following ways:

getDFSMessage()
Returns the message segments concatenated together as a single string.

getDFSMessageSegments()
Returns a vector of the segments of the message.

In addition, the individual fields of the message segments (DFSLL1, DFSZZ1, DFSDATA1, etc.) can be accessed by a Java application or servlet.

Any of the methods above can be used by Java applications. Currently, however, WebSphere Studio does not allow selection of a method for use in displaying data on a servlet's output template. In order for a servlet to display a DFS message, the individual fields of the DFSMsg object should be selected (DFSLL1, DFSZZ1, DFSDATA1, etc.) while in WebSphere Studio.

A composite bean that uses IMS Connector for Java should include a DFSMsg bean as well as bean(s) representing the output of the IMS transaction. VisualAge for Java's Enterprise Access Builder populates the appropriate output bean, based on the data that is returned by IMS.

Attention: Attention: The **DFSMO1A** and **DFSMO1B** beans of IMS Connector for Java 1.1.0 have been deleted in this release. You must replace these beans with the DFSMsg bean.

Related Reading: For more information on displaying a DFS message with a servlet, see Building a Java Servlet to Run an IMS Transaction.

IMS Messages

The messages sent to IMS or received from IMS by a Java application or servlet using IMS Connector for Java can be any of the following:

- IMS transaction input messages
- IMS transaction output messages
- IMS status or error messages (also called "DFS" messages)
- IMS commands
- IMS command output messages

IMS Connector for Java is primarily designed to handle the first three types of messages. IMS messages, both input and output, can consist of multiple segments. Building a Java Application for an IMS Transaction with Multi-Segment Output Messages and Building a Java Application for an IMS Transaction with Multi-Segment Input Messages provide examples of how to create Java applications that run IMS transactions with multiple segment messages. For a transaction input or output message, no restrictions exist on the number of segments. The IMS Connector for Java-supplied bean, DFSMsg, supports both single-segment IMS output messages and multiple-segment IMS output messages of 22 segments or less. These ″DFS″ output messages are sent by IMS to reflect IMS errors or status changes in response to a transaction request received by IMS. They are also returned as the output response from an IMS command. While it is possible to submit a command to IMS from a Java application or servlet using IMS Connector for Java that generates more than 22 segments of command output, that Java application or servlet is restricted to receiving a maximum of the first 22 segments of that command output.

IMS Message Format

Message segments that are sent to and received from IMS transactions always begin with a 2-byte segment length field (called **LL**), followed by a 2-byte field that contains IMS information (called **ZZ**). The 2-byte segment length field represents the length of the entire message segment, including the **LL** and **ZZ** fields. The data of the message segment follows the **LL** and **ZZ** fields. In the case of the first segment of the transaction's input message, up to the first the first n+1 bytes of the data portion of the segment contain the n-byte transaction code, followed by a blank.

COBOL definitions that represent transaction input and output message segments must contain fields for the segment length, the IMS information, and the data portions of a segment. A Java application program or servlet must "set" the individual fields of a transaction input message segment before sending that segment to IMS. The fields are set by doing the following:

- Set the individual fields of the data portion of the message segment using information that is provided by the end user.
- Set the segment length field (**LL**) to be the sum of the defined lengths of all the data fields, plus 4. If **LL** is not set correctly, IMS Connector for Java might throw an exception or unpredictable results might occur, depending upon the IMS application program. Consider adding a method to your EAB command that uses the getSize() method of com.ibm.record.CustomRecord to return the size of your transaction input message.
- Setting the IMS information field (**ZZ**) is optional.

IMS Logon Information

The IMS logon information (user ID, password, and group name) provided to the run-time context of a Java application or servlet is used by IMS Connector for Java as follows:

The user ID, password, and group name are placed in the OTMA message sent by IMS Connector for Java to the host component, IMS Connect. IMS Connect then calls the host's Security Authorization Facility (SAF) under control of the IMS Connect **SETRACF** command:

- If the **SETRACF ON** command is issued, the SAF is called. If user authentication succeeds, the UTOKEN built by the SAF is passed on to IMS for authorization. If user authentication fails, IMS Connect returns an error to IMS Connector for Java.
- If the **SETRACF OFF** command is issued, the SAF is not called by IMS Connect. The user ID and group name are passed on to IMS for authorization.

Four levels of authorization are available in IMS. These levels are controlled by the IMS command, **/SECURE OTMA**, as follows:

1. If **/SEC OTMA NONE** is issued, no authorization is performed.
2. If **/SEC OTMA CHECK** is issued, the UTOKEN or user ID and group name are passed to the SAF to ensure the user (in the group) is authorized to execute the transaction (or command).
3. If **/SEC OTMA FULL** is issued, in addition to performing CHECK level authorization, a user security profile is built in the MPP region for use by non-IMS services.
4. If **/SEC OTMA PROFILE** is issued, the result is the same as if **/SEC OTMA FULL** is issued, because IMS Connector for Java sets the Security Data section of the OTMA message prefix for each transaction to the **FULL** option.

Relationship of IMS logon information to EAB commands
IMS logon information is set in the run-time context of a Java application or servlet. For example:

```
com.ibm.connector.imstoc.IMSLogonInfoItems logon =
  new com.ibm.connector.imstoc.IMSLogonInfoItems(
    runtimeContext.getLogonInfo(),cmd.getConnectionSpec());
logon.setUser("yourUID");
logon.setPassword("yourPwd");
logon.setUser("yourGrp");
```

Typically this is performed at the beginning of an application or servlet, and lasts for the duration of the application or servlet. However, in some situations, it might be desirable to change the IMS logon information within an application or servlet. For example, an application or servlet might run multiple transactions that belong to different groups. In this case, the setGroup() method must be invoked before each

execute of an EAB command in order to set the appropriate group name for the transaction that the command runs. This applies to both of the following cases:

- A different EAB command is used for each transaction
- The same EAB command is used for all the transactions

In certain situations the same EAB command can be used for multiple transactions. For example, multiple transactions might have the same input and output message structure but different transaction codes.

Attention: The IMSLogonInfo class of IMS Connector for Java 1.1.0 has been deleted in this release. You must replace this class with class com.ibm.connector.imstoc.IMSLogonInfoItems.

Usability enhancement
In IMS Connector for Java 1.2.0 and later releases, the IMS logon information cannot be changed between invocations of the execute() method of the same EAB command. In IMS Connector for Java 1.2.0, the IMS logon information can be changed between invocations of the execute() method for the same EAB command. This is useful if, for example, the same EAB command is used for different transactions, and the transactions have different group names.

Synchronization Level

The synchronization level can be specified for an EAB command that runs an IMS transaction. Synchronization level determines the way in which IMS and the client that contains the EAB command interact. An EAB command that runs an IMS transaction can execute with a synchronization of level of `None` or `Confirm`.

Currently, all IMS Connector for Java interactions use the OTMA protocol **Commit Mode 1,** also referred to as send-then-commit. Under this protocol, if the synchronization level is Confirm, IMS sends the output message to the client and then waits for a response from the client. It is the responsibility of the client to respond to IMS. If the synchronization level is None, IMS commits any changes made by the IMS application without waiting for a response from the client.

Synchronization level none
When an EAB command uses a synchronization level of `None`, the transaction runs and IMS sends the output message to the client. Any database changes are then committed. IMS does not require that the client send a message in response to the transaction output message in order to commit the database changes. Synchronization level `None` is typically used for Java applications and servlets that run IMS transactions that browse or query host databases.

Synchronization level confirm
When an EAB command uses the synchronization level Confirm, the transaction runs and IMS sends the output message to the client without committing any database changes. IMS does not complete the transaction until the client responds to the transaction output message by sending a positive or negative acknowledgment to IMS. If the client is satisfied with the transaction output, it responds by sending a positive acknowledgment. IMS then completes the transaction by committing the database changes, if necessary. If the client is not satisfied with the transaction output (or does not want to continue with the transaction for any reason) it responds by sending a negative acknowledgment. IMS then rolls back any changes to the database.

If IMS Connect on the host encounters any errors from the Java application, it can also send a negative acknowledgment to IMS to abort the transaction. For example, this might occur if communication with the Java application and IMS Connect is lost while processing the transaction.

The client application can respond to IMS in one of two ways.

1. If the coordinator object obtained from the current runtime context is com.ibm.connector.infrastructure.NullCoordinator, the client sends a positive or negative acknowledgement to IMS. This is done by first creating an EAB command with interaction mode MODE_ACK (for a positive acknowledgment) or MODE_NACK (for a negative acknowledgment), then executing the appropriate EAB command to send the acknowledgement message to IMS.
2. If the coordinator object obtained from the current runtime context is not NullCoordinator, and is instead com.ibm.connector.infrastructure.java.JavaCoordinator, for example, then the client invokes the commit() or rollback() method on the coordinator object, following receipt of the output message. The implementation of these methods will, in turn, send the MODE_NACK acknowledgement message to IMS.

In the case of an IMS conversation, different synchronization levels may be used for one or more of the iterations of the conversation. Since the conversational programming model presented by IMS Connector uses a new instance of the EAB command for each iteration of the conversation, the synchronization level must be explicitly set for each iteration. Otherwise, the synchronization level set for the command at development time (usually the default, SYNC_LEVEL_NONE) will be used for the iteration.

If SYNC_LEVEL_CONFIRM is specified for a given iteration of an IMS conversation, either the commit() or rollback() method of JavaCoordinator must be executed following receipt of the output message for that iteration of the conversation. These methods submit the appropriate MODE_ACK or MODE_NACK messages to IMS. MODE_ACK or MODE_NACK messages can also be sent to IMS by creating two additional EAB commands which would be used in a NullCoordinator environment. However, it is recommended that applications use the simpler JavaCoordinator interface. See Building an Application to Run an IMS Transaction with Synchronization Level Confirm for an illustration of an application that uses an IMS nonconversational transaction with SYNC_LEVEL_CONFIRM. Iterations of IMS conversations that use SYNC_LEVEL_CONFIRM would be handled in a similar fashion, as far as commit and rollback processing is concerned.

Most likely the synchronization level will be the same for all iterations of the IMS conversation, in which case it would be appropriate to set it in the IMSInteractionSpec when the EAB command is developed.

A typical application scenario might be

1. The application invokes a transaction EAB command with interaction mode MODE_SEND_RECEIVE and the synchronization level is SYNC_LEVEL_CONFIRM.
2. The application receives the transaction output from IMS.
3. The application performs some logic (or interacts with the end-user) to decide whether the transaction should be completed.
4. Based on the results from the previous step, the application invokes the commit() or rollback() methods of JavaCoordinator to respond to IMS.

5. IMS completes the transaction by committing or rolling back the database changes, based on the type of response (positive or negative acknowledgment) that it received.

Depending on the coordinator used with the run-time context, the client can execute an EAB command with interaction mode MODE_ACK or MODE_NACK instead of invoking commit() or rollback() of JavaCoordinator. Because the MODE_ACK or MODE_NACK EAB command only sends acknowledgment messages to IMS, it does not contain any input or output beans.

Related Reading: For more information on building an application to run an IMS transaction with synchronization level, see Building and Application to Run an IMS Transaction with Synchronization Level Confirm.

Connection Management

Connection pooling is key to enhancing the performance of Java applications or servlets that access IMS transactions. This connection management feature is provided by the IBM Common Connector Framework (CCF). CCF is a set of Java APIs that provides infrastructure services like connection management, transaction services, security, and tracing facilities to Java applications and servlets. CCF currently provides two types of connection manager:

com.ibm.connector.internal.DefaultConnectionManager
: DefaultConnectionManager is the default connection manager used by the RuntimeContext classes (i.e. RuntimeContext and JavaRuntimeContext) shipped with VisualAge for Java. It does not support connection pooling. A new connection is created for each transaction request and the connection is disconnected at the end of the execution.

com.ibm.connector.connectionmanager.ConnectionManager
: ConnectionManager is a 'real' connection manager that supports connection pooling. It maintains a pool of reusable connection for each port that IMS Connect uses. If you have multiple IMS Connect components running, the ConnectionManager will maintain a separate pool for each port on each host (that is, each Hostname/Port number combination). When the ConnectionManager class is used with IMS Connector for Java, a connection with IMS Connect is obtained for each transaction request, and the connection is returned back to the pool to be reused by subsequent requests. Using persistent socket connections minimizes the overhead of socket initialization, and significantly enhances performance.

There should be one instance of the ConnectionManager class used within a component server (for example, WebSphere Application Server). If your component server supports the CCF infrastructure, a single instance connection manager would be established and be accessed by the servlets via IMS Connector for Java. If your component server does not support the CCF infrastructure, you can use a servlet to store a single instance of the ConnectionManager class. This single instance can be accessed by any servlet by setting it on the RuntimeContext of a Java servlet.

As a result, to ensure connection pooling is established properly in your component server for running your servlet requests with CCF connectors, there should be one instance of connection manager AND the type of connection manager being used should support connection pooling (for example, the type com.ibm.connector.connectionmanager.ConnectionManager).

IMS Connector for Java has provided a sample implementation of a servlet that stores a single global connection manager of type com.ibm.connector.connectionmanager.ConnectionManager in the ServletContext. The source code of the servlet is in the RegisterConnectionManager servlet in the com.ibm.connector.ims.sample.cm package included in the IMS Connector Sample Feature. This sample could be used when your component server does not support the CCF infrastructure or it does not use a type of connection manager that supports connection pooling. This sample servlet must be loaded once when the component server starts to establish the connection manager. To load the sample servlet, invoke the following URL from your web browser:

http://*serverhostname*/servlet/com.ibm.connector.ims.sample.cm.RegisterConnectionManager

where *serverhostname* is the hostname of your component server.

To access the global instance of the ConnectionManager from the ServletContext in your servlet code, add the following sample code in the method that uses executes the EAB command (For example, for a servlet generated by WebSphere Studio, it will be placed in the performTask() method). This code should be placed before the execute() method of the EAB command is called:

```
// Access the global instance of ConnectionManager from the ServletContext.
ServletConfig sc = getServletConfig();
ServletContext sctx = sc.getServletContext();
com.ibm.connector.connectionmanager.ConnectionManager connMgr =
     (com.ibm.connector.connectionmanager.ConnectionManager)
     sctx.getAttribute("CCFConnectionManager);
// Setup the Connection Manager with the RuntimeContext
RuntimeContext.getCurrent().setConnectionManager(connMgr);
```

A sample usage of the above code can be found in the servlet examples included in the IMS INSTALL/IVP Sample Application.

IMS Conversations

IMS Connector for Java Conversational Support allows customers to build Java applications and Web applications to access IMS conversational transactions.

A conversational IMS transaction is a transaction that is defined to IMS as being conversational, meaning a that it can process transactions made up of several individual steps. An IMS transaction is made up of a connected series of client-to-program-to-client interactions. The IMS conversational program receives messages from the client, processes the requests and replies to the client. It also saves the intermediate data from the transaction in the scratch pad area (SPA). When the user enters more data from the client, the program has the data it saved from the last message in the SPA, and thus can continue processing the request without having the user enter the data again. When the client sends a message to initiate the next iteration of the conversation, the program uses the data in the new message along with the data it saved in the SPA at the end of the last iteration of the conversation as its input. In more complex conversations, different transactions can be invoked using an immediate or deferred program switch.

Related Reading: For more information on program switching, refer to the *IMS Conversations* section in the *Application Programming: Transaction Manager* volume of the IMS books.

In IMS Connector for Java Conversational Support, the client participating in an IMS conversation transaction is typically a Web application. In a Web application the

user submits a series of requests that invoke one or more Java servlets which in turn process the iterations of an IMS conversation. The Java servlet receives an input request from the browser and utilizes...to send a conversational transaction request to IMS via IMS Connect. IMS Connect then forwards the transaction request to IMS through OTMA. IMS schedules the conversational transaction which starts a new IMS conversation. The IMS application processes the request and sends the output back to the Java servlet via IMS Connect and IMS Connector for Java. The Java servlet then loads the appropriate JavaServer Page to display the output of that iteration to the user in the browser.

IMS Connector for Java Conversational Support provides two programming models from which to build a conversational Web application. The programming models are the Conversational HttpSession Model and the Conversational Navigator Model. The following commands, however, are not supported for IMS conversations that are initiated using IMS Connector for Java Conversational Support:

```
/EXIT
/HOLD
/RELEASE
```

Chapter 2. Preparing to use IMS Connector for Java

This topic describes how to create an environment in which you can develop and run Java programs that use IMS Connector for Java. It provides installation and configuration information for the following products:

- VisualAge for Java Enterprise Edition Version 3.5
- WebSphere Studio Version 3.5
- WebSphere Application Server Version 3.5

This section also provides verification procedures for using IMS Connector for Java within VisualAge for Java and WebSphere Application Server. Use these procedures to verify that the installation of IMS Connector for Java with these products is complete.

These three products provide an integrated environment that you can use to develop and run Java application programs and servlets that access IMS transactions. The IMS Connector for Java is included in VisualAge for Java.

Preparing your VisualAge for Java Environment

VisualAge for Java Enterprise Edition 3.5 can be used to both develop and run Java programs that use IMS Connector for Java to access IMS transactions. This section describes how you prepare your VisualAge for Java environment.

Configuring VisualAge for Java

IMS Connector for Java is a feature of VisualAge for Java Enterprise Edition 3.5 that is added after installation of VisualAge for Java has completed; it does not require a separate installation process. Because IMS Connector for Java is part of VisualAge for Java, its software prerequisites are the same as those for VisualAge for Java.

After you add the IMS Connector for Java feature to your VisualAge for Java installation, you have access to the following:

- IMS Connector for Java

  IMS Connector for Java is a class library that consists of two packages: **com.ibm.connector.imstoc** and **com.ibm.imstoc**. All of the classes in the **com.ibm.imstoc** package, and many of the classes in the **com.ibm.connector.imstoc** package are "support" classes, and are not used by VisualAge for Java developers. The classes and methods that are used in the VisualAge for Java application development process are documented in the Reference section of the VisualAge for Java web help. To view this documentation, click:

  Help —>Reference —> IBM APIs —> Connectors —> IMS Connect
- *IMS Connector for Java User's Guide and Reference* in HTML and PDF formats. For information on viewing or printing the PDF format of the *User's Guide and Reference*, see the Visual Age of Java web help and click on **Help —>PDF Index** in any window of the VisualAge for Java IDE. To access the HTML version of these documents, bring up the VisualAge for Java web help by clicking on **Help—>Help Home Page** and then clicking on:

Topics —>Tasks —> Accessing the Enterprise —> Accessing Transactions with the IMS Connector for Java

© Copyright IBM Corp. 1998, 2000

- IMS Connector for Java IVP program that you can use to verify that a Java application using IMS Connector for Java can access IMS OTMA from within the VisualAge for Java environment.
- IMS Connector for Java IVP servlet that you can use to verify that a servlet using IMS Connector for Java can access IMS OTMA from within a WebSphere Application Server environment.
- Sample code to aid in the development of your IMS Connector Java applications.
- IMS Connector for Java Samples

Important: The IMS Connector for Java IVP servlet is currently provided only with VisualAge for Java. For instructions on how to run this program in the WebSphere Application Server environment, see *Using the IMS Connector for Java IVP to Verify Your WebSphere Application Server Environment.*

To develop Java code that uses IMS Connector for Java, you must first configure your VisualAge for Java workspace by adding the following class libraries:

- IBM Enterprise Access Builder Library
- Connector IMS Connect
- IBM Common Connector Framework
- IBM Java Record Library

To add these class libraries to your workspace, you must add one of the following features:

- IMS Connector
- IMS Connector Samples

The IMS Connector feature includes all of the classes needed to use IMS Connector for Java either directly or through internal prerequisites which are brought in automatically if they do not already exist in your workspace. The IMS Connector feature itself is a prerequisite to the IMS Connector Samples feature which includes all of the IMS Connector for Java sample packages. As a result, you would install the IMS Connector feature if you do not want to install the sample packages and IMS Connector Samples if you want to install IMS Connector for Java along with all of the IMS Connector for Java sample packages. If, at any point in the future, you choose to install the sample packages, you simply install the IMS Connector Samples feature at that time. All of the prerequisites will have already been installed.

There are several programming example sections in this documentation which provide step-by-step instructions for building various types of applications that utilize IMS Connector for Java. There are corresponding samples which are installed with the IMS Connector Samples feature. Due to the fact that there is a difference between the project and package names used in the samples and those used in the programming examples section of this documentation, you will be able to compare your work in creating the examples against the corresponding sample code provided. Use the following procedure to add one of these features:

1. Start the Workbench in the VisualAge for Java IDE:

   Click the **Start** icon and click **Programs —> IBM VisualAge for Java for Windows —> IBM VisualAge for Java**. If the VisualAge **Welcome to VisualAge** dialog box appears, click **Go to the Workbench** and click **OK**. The IDE workbench appears.
2. From the **File** menu, click **Quick Start**.
3. In the **Quick Start** window, click **Features** on the left pane. Then, select **Add Feature** on the right pane and click **OK**.

4. Select one of the class libraries from the features list and click **OK**. The class library named **Connector IMS Connect** corresponds to the IMS Connector for Java feature.
5. At this point you have completed the installation of IMS Connector for Java. You may repeat the previous step to add additional class libraries to your workspace, if you choose to do so. You can add multiple features at one time by pressing the **Ctrl** key and selecting the features using the mouse.

Normally, problems are reported as each class library is added. These problems are resolved when you import the final class library.

Using the IMS Connector for Java IVP to Verify Your VisualAge for Java Environment

The IMS Connector for Java Installation Verification Program (IVP) is a Java application that is included with VisualAge for Java. You can use the IVP to verify that a Java application that uses IMS Connector for Java can run within VisualAge for Java and can successfully access the target IMS OTMA environment. The IMS Connector for Java IVP sample program is contained in the com.ibm.connector.ims.sample.ivp package in the Connector IMS Samples project that is installed as part of the Connector IMS Samples feature.

The IVP verifies whether a Java application running within VisualAge for Java can access IMS OTMA.

To run the IVP, verify that the following components are running on your target host machine:

- IMS Connect, Version 2.1.3 or higher

  Ensure that the outstanding IMS Connect reply appears on the target machine's system console. For example:

  ```
  14.22.25 STC00159 *13 HWSC0000I *IMS CONNECT READY*  ICONNAME
  ```

  Verify that the PORT and DATASTORE are ACTIVE by entering the HWS command VIEWHWS at the HWS outstanding reply.

  Verify that the eyecatcher in module HWSHWS00 in the HWS RESLIB is at least:

  ```
  IMS Connect Version 02.1.3
  ```

- IMS 5.1 or higher

  Ensure that the IMS outstanding reply appears on the target machine's system console. For example:

  ```
  14.26.55 JOB00175 *15 DFS996I *IMS READY*   SYS3
  ```

  Verify that the XCF status of both the IMS and IMS Connect members is ACTIVE by entering the IMS command /DISPLAY OTMA at the outstanding IMS reply:

  ```
  14.27.39 JOB00175 DFS000I GROUP/MEMBER XCF-STATUS USER-STATUS
  SECURITY SYS3
  14.27.39 JOB00175 DFS000I -IMSMEM ACTIVE SERVER
  NONE SYS3
  14.27.39 JOB00175 DFS000I -ICONNMEM ACTIVE ACCEPT
  TRAFFIC SYS3
  ```

In addition, ensure that TCP/IP can be used to communicate between the Java application and the IMS Connect host component by verifying that you can successfully "ping" the target host machine from your VisualAge for Java workstation.

The IVP does not require a host IMS application.

Important: Service Pack 3 (or higher) must be applied to your Windows NT 4.0 VisualAge for Java environment to successfully run the IMS Connector for Java IVP. If you attempt to run the IVP in a Windows NT environment without Service Pack 3, a Java exception is thrown that is very difficult to diagnose.

Running the IVP

Update VisualAge for Java's class path to point to the locations of the required class libraries. Do the following:

1. Select the IMSConnIVPMainDialog class in package com.ibm.connector.ims.sample.ivp.
2. Ensure that the **Class path** tab is selected on the **Properties for IMSConnIVPMainDialog** window and click **Compute now** to compute the class path for the IVP. Click **OK**.

To run the IVP, select the IMSConnIVPMainDialog class in package **com.ibm.connector.ims.sample.ivp**. From the **Selected** menu, click **Run —> Run main...** The IMS Connector for Java IVP window appears.

Because execution of the IVP causes the IMS **/STA OTMA** command to be sent to the target IMS machine, you must provide values for the fields described below before clicking **SUBMIT**.

Figure 4. The VisualAge for Java IVP GUI

Host name
The TCP/IP host name of the machine running IMS Connect.

Port The port assigned to IMS Connect.

Datastore
The Datastore ID you specified when you installed IMS Connect.

User ID
The SAF user ID to be passed to OTMA. For example, the RACF user ID. User ID may be optional depending on how you specify IMS OTMA and IMS Connect security. However, you can provide it, along with a password, if you want to verify your installation's security configuration.

Password
Password may be optional (see User ID).

Group name
Group name may be optional (see User ID).

The objective of the IVP is to receive a message from IMS in reply to the **/STA OTMA** command. One of two possible messages is returned, depending on the security configuration of your host environment and whether the given user ID is permitted to issue the **/STA** command. Either message indicates successful execution of the IVP:

- DFS1292E SECURITY VIOLATION
- DFS058I 17:33:32 START COMMAND COMPLETED

This test does not verify that a particular IMS transaction can run, only that the path to IMS is available.

If an exception is returned in response to the command, check the console window. The exception and stack trace provide diagnostic information. The following are common causes for IVP failure:

- Host name is misspelled or is not sufficiently qualified
- Incorrect port
- TCP/IP failure. Always ensure a successful "ping" prior to running the IVP
- Datastore name is incorrect or is misspelled. Datastore name must be in uppercase characters.
- Wrong level of IMS Connect running on the host. Ensure that IMS Connect is V2.1.3 or above.

Important: If you do not compute the class path for class IMSConnIVPMainDialog as described above, the IMS Connector for Java IVP throws an exception, indicating that one of the classes cannot be found. In some cases, the IVP might not respond.

Preparing your WebSphere Studio Environment

You can use WebSphere Studio to develop servlets that access IMS transactions. These servlets are generated from Enterprise Access Builder (EAB) commands that use IMS Connector for Java. This section describes what you need to do to prepare for using these commands in the WebSphere Studio environment.

Installing Class Libraries

Attention: If you have both VisualAge for Java 3.5 and WebSphere Studio 3.5 installed on the same machine, you do not need to complete the steps described in this section. However, you must modify your WebSphere Studio's CLASSPATH environment variable to point to the class libraries being installed by VisualAge for Java as follows:

1. In WebSphere Studio, select **Tools —> Preferences** from the menu bar.
2. Click on the **Java** tab and then enter the following in the **Classpath** field:

```
{StudioClasspath};<VAJ_Install_dir>\eab\runtime30;
```

IMS Connector for Java is not currently packaged with WebSphere Studio. To generate a servlet from an EAB command that uses IMS Connector for Java, you need the following class libraries installed on the WebSphere Studio workstation:

- IBM Common Connector Framework
- IBM Enterprise Access Builder Library
- IBM Java Record Library

- Connector IMS

The following steps show you where to obtain the class libraries (.jar files) and how to set up the CLASSPATH environment variable to point to the .jar files. WebSphere Studio generates the servlet using the EAB command and these class libraries.

In this section, the parameters represent the following:

<VAJ_install_dir>
The VisualAge for Java installation directory. For example, d:\IBMVJava.

<IBM_Connectors_install_dir>
The IBM Connectors installation directory. For example, d:\IBM Connectors. Initially, *<IBM_Connectors_install_dir>* is **<vaj_install_drive>\IBM Connectors**, where <vaj_install_drive> is the drive on which VisualAge for Java is installed.

<target_dir>
The path on your WebSphere Studio workstation where the .jar files are placed. For example, c:\imsconn.

1. Copy the following class libraries (.jar files) from their directories (indicated below) to your WebSphere Studio target directory (*<target_dir>*):
   - IBM Common Connector Framework
     File ccf.jar is located in the *<IBM_Connectors_install_dir>*\classes directory.
   - IBM Enterprise Access Builder Library
     File eablib.jar is located in the *<VAJ_Install_dir>*\eab\runtime30 directory.
   - IBM Java Record Library
     File recjava.jar is located in the *<VAJ_Install_dir>*\eab\runtime30 directory.
   - IMS Connector for Java
     File imsconn.jar is located in the *<IBM_Connectors_install_dir>*\classes directory.
2. Modify (or append to) your WebSphere Studio's CLASSPATH environment variable for WebSphere Studio to point to the above .jar files.

   Modify the CLASSPATH environment variable in WebSphere Studio by clicking **Tools —> Preference**. Click on the **Java** tab and enter the following in the **Classpath** field

   ```
   {StudioClasspath};<target_dir>\ccf.jar;<target_dir>\eablib.jar;
   <target_dir>\recjava.jar;<target_dir>\imsconn.jar;
   ```

Preparing your WebSphere Application Server Environment

You can use WebSphere Application Server to run servlets that use IMS Connector for Java to access IMS transactions. These servlets are typically generated using WebSphere Studio. This section describes what you need to do to prepare for using these servlets in the WebSphere Application Server environment.

Important: IBM WebSphere Application Server is installed as a plug-in to your Web server. WebSphere Application Server must be installed after your Web server is installed. If you reinstall your Web server after installing WebSphere Application Server, be sure to reinstall WebSphere Application Server. Otherwise, WebSphere Application Server cannot be started.

Installing class libraries

The IMS Connector for Java is not currently packaged with all editions of WebSphere Application Server. To run a servlet that is generated from WebSphere

Studio (and is created using an EAB command that uses IMS Connector for Java) in WebSphere Application Server Standard and Advanced Edition, you need the following class libraries installed on the WebSphere Application Server workstation:

- IBM Common Connector Framework
- IBM Enterprise Access Builder Library
- IBM Java Record Library
- Connector IMS

In this section, the parameters represent the following:

- *<install_dir>*

  The VisualAge for Java installation directory. For example, d:\IBMVJava (on Windows), or /usr/WebSphere/AppServer/classes (on AIX).

- *<IBM_Connectors_install_dir>*

  The IBM Connectors installation directory. For example, d:\IBM Connectors. Initially, *<IBM_Connectors_install_dir>* is **<vaj_install_drive>\IBM Connectors**, where <vaj_install_drive> is the drive on which VisualAge for Java is installed.

- *<target_dir>*

  The path on your Web server's workstation where the .jar files are placed. For example, d:\WebSphere\AppServer\classes or /usr/WebSphere/AppServer/classes.

- *<websphere_app_server_install_dir>*

  The WebSphere Application Server installation directory. For example, d:\WebSphere\AppServer (on Windows), or /usr/WebSphere/AppServer (on AIX).

The following steps show you where to obtain the class libraries (.jar files) and how to set them up for use by the WebSphere Application Server:

1. Copy the following class libraries (.jar files) from the directories indicated below to the target directory of your Web server:
   - IBM Common Connector Framework
     File ccf.jar is located in the directory *<IBM_Connectors_install_dir>*\classes.
   - IBM Enterprise Access Builder Library
     File eablib.jar is located in the directory *<install_dir>*\eab\runtime30.
   - IBM Java Record Library
     File recjava.jar is located in the directory *<install_dir>*\eab\runtime30.
   - IMS Connector for Java
     File imsconn.jar is located in the directory *<IBM_Connectors_install_dir>*\classes.
   - Target Directory
     The target directory is the directory on your Web server's workstation where the .jar files are placed. For example, d:\WebSphere\AppServer\classes (on Windows) or /usr/WebSphere/AppServer/classes (on AIX).
2. Modify the CLASSPATH environment variable for WebSphere Application Server to point to the class libraries (.jar files) listed above.

   Add the paths of the above .jar files to the WebSphere Application Server classpath by doing the following:

   a. In WebSphere Application Server Version 3.5's **Administrative Console**, select **View -> Topology** from the menu bar. Click the **+** sign to expand the view of the **WebSphere Administrative Domain** node.

b. Click on the + sign again to expand the view of the node that has the TCP/IP host name of your web server machine. Select the application server of the node; for example, Default Server.

c. On the **General** tab, append the following to the **Command Line Arguments** field.
For example, **on Windows:**

```
-classpath <target_dir>\ccf.jar;<target_dir>\eablib.jar;<target_dir>\recjava.jar;<target_dir>
```

For example, **on AIX:**

```
-classpath <target_dir>/ccf.jar:<target_dir>/eablib.jar:<target_dir>/recjava.jar:<target_dir>
```

d. Restart the application server.

Using the IMS Connector for Java IVP to verify your WebSphere Application Server environment

The IMS Connector for Java Installation Verification Program (IVP) is a Java servlet that is currently included with VisualAge for Java.

You can use the IVP servlet to verify that a Java servlet that uses IMS Connector for Java can successfully access the target IMS OTMA from the WebSphere Application Server environment.

Important: This program is currently provided only with VisualAge for Java. It is not currently provided with WebSphere Application Server.

Prerequisites for running the IVP servlet
Before you run the IVP servlet, ensure that the following components are running on your target host machine:

- IMS Connect
  - Ensure that the HWS outstanding reply appears on the target machine's system console. For example:

    ```
    *16.59.05 STC00187 *57 HWSC0000I *IMS CONNECT*  HWS_NAME
    ```

    Verify that the PORT and DATASTORE are ACTIVE by entering the HWS command VIEWHWS at the HWS outstanding reply.
  - RESLIB contains at least '0220' as the version number. For example,

    ```
    HWSHWS00+0220+....
    ```
- IMS 5.1 or higher
  Ensure that the IMS outstanding reply appears on the target machine's system console. For example:

  ```
  *14.23.55 JOB00175 *60 DFS996I *IMS READY*  SYS3
  ```

  Verify that the IMS Connect Connection member is ACTIVE by entering the IMS command /DISPLAY OTMA at the IMS outstanding reply.

  For information on using other releases of IMS or for using IMS TCP/IP OTMA Connection, see Prerequisites for Using IMS Connector for Java and the documentation for earlier releases of IMS Connector for Java.

In addition, ensure that TCP/IP is running on your target host machine by verifying that you can successfully "ping" the target host machine from the Web server machine.

The IVP servlet does not require a host IMS application.

Deploying the IVP

The IVP consists of the following files:

- IMSConnIVPServletHTMLInput.html
- IMSConnIVPServletHTMLResults.jsp
- IMSConnIVPServletHTMLError.jsp
- IMSConnIVPServlet.class
- IMSConnIVPServlet.servlet
- IMSConnIVPServlet.jar
- Master.css

To deploy the IVP servlet, do the following:

- Start WebSphere Application Server. You must first start the web server, and then start WebSphere Application Server separately.

  Using Windows NT

  1. To start the web server for an IBM HTTP server, for example, you do the following::
     a. Click **Start—>Settings—>Control Panel.**
     b. Click the **Services** icon. Select **IBM HTTP Server** and click **Start**.
     c. Click **Close** to close the window.
  2. To start the WebSphere Application Server, do the following:
     a. Click **Start—>Settings—>Control Panel**.
     b. Click the **Services** icon. Select **IBM WS AdminServer** and click **Start**.
     c. Click **Close** to close the window.
     d. Click **Start—>Programs—>IBM WebSphere—>Application Server V3.5—>Administrator's Console.**
     e. Ensure that the **Topology** view is displayed. To switch to the **Topology** view, choose **View—>Topology** from the menu bar. Click the **+** sign to expand the view of the **WebSphere Administrative Domain** node.
     f. Click on the **+** sign again to expand the view of the node that has the TCP/IP host name of your Web server machine.
     g. Right click on **Default Server** under the host node in the previous step, and select **Start**. A dialog box displaying "Command 'DefaultServer.start' completed successfully" will appear when the server has been started. Click **OK** on the dialog box.

  Using AIX

  1. To start the web server, enter one of the following:
     a. For a Domino Go server, enter: startsrc -s httpd
     b. For an IBM HTTP server, enter: apachectl start
  2. To start the WebSphere Application server, do the following:
     a. From the AIX command line, invoke: /usr/WebSphere/AppServer/bin/admin.sh
     b. Ensure that the **Topology** view is displayed. To switch to the **Topology** view, choose **View—>Topology** from the menu bar. Click the **+** sign to expand the view of the **WebSphere Administrative Domain** node.
     c. Click on the + sign again to expand the view of the node that has the TCP/IP host name of your Web server machine.

d. Right click on **Default Server** under the host node in the previous step, and select **Start**. A dialog box displaying ″Command 'DefaultServer.start' completed successfully″ will appear when the server has been started. Click **OK** on the dialog box.

- Ensure that you have prepared your WebSphere Application Server to use IMS Connector for Java, as described in Preparing Your WebSphere Application Server Environment.

  Copy the IVP files to the proper Web Application Server directories:

  1. Copy the HTML and .jsp files from the **<***IBM_Connectors_install_dir*>\imsconn\samples\servlets\ivpwas35 directory to the document root directory of your Web server; e.g. (on Windows) d:\IBM HTTP Server\htdocs.
  2. Copy the theme folder and its contents from the <*ibm_connectors_install_dir*>\imsconn\samples\servlets\ivpwas35 directory to the document root of your Web server.
  3. Copy the com folder and all its subfolders containing the .servlet and .class files from the <*ibm_connectors_install_dir*>\imsconn\samples\servlets\ivpwas35 directory to the default_app Web application servlet directory. Make sure you preserve the Java package directories of these files; e.g., copy the .servlet and .class files to (on Windows) <*websphere_install_dir*>\host\default_host\default_app\servlets\com\ibm\connector\ims\sample\
  4. Copy the IMSConnIVPServlet.jar file from <*ibm_connectors_install_dir*>\imsconn\samples\servlets\ivpwas35 directory to the default_app servlet directory; e.g., (on Windows) <*websphere_install_dir*>\hosts\default_host\default_app\servlets.

  After all the files are copied, you should have, for example, the following directories (bold) and directory contents:

  – **WebSphere**
    - **AppServer**
      • **hosts**
        – **default_host**
          - **default_app**
            • **servlets**
              – IMSConnIVPServlet.jar
              – **com**

  Under the **com** directory, you have the following directories and files:

  – **ibm**
    - **connector**
      • **ims**
        – **sample**
          - **ivp**
            • **was35**
              – **servlet**
                - IMSConnIVPServlet.servlet
                - IMSConnIVPServlet.class

In addition, the theme folder and its contents and all the HTML and JSP pages should be in the document root of your Web server. For example:

- **IBM HTTP Server**
  - **htdocs**
    - IMSConnIVPServletHTMLInput.html
    - IMSConnIVPServletHTMLResults.jsp
    - IMSConnIVPServletHTMLError.jsp
    - **theme**
      - Master.css

Important: In the scenario outlined above, the following parameters are defined as described below:

- *<IBM_Connectors_install_dir>*

  The IBM Connectors installation directory. For example, d:\IBM Connectors. Initially, *<IBM_Connectors_install_dir>* is **<vaj_install_drive>\IBM Connectors**, where <vaj_install_drive> is the drive on which VisualAge for Java is installed.
- *<websphere_install_dir>*

  The WebSphere Application Server installation directory. For example, d:\WebSphere\AppServer (on Windows) or /usr/WebSphere/AppServer (on AIX).
- *<servername>*

  The name of your Web server machine.

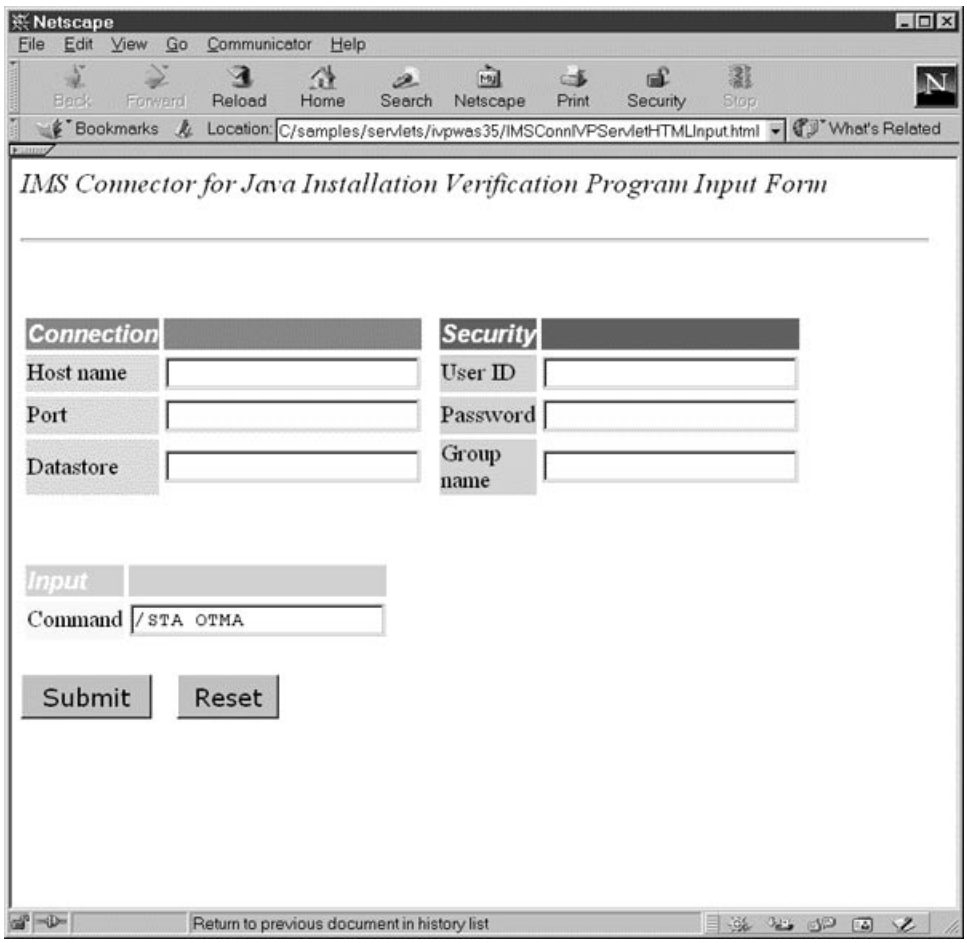
Running the IVP Servlet

1. From a Web browser, enter the following URL:

   http://*<servername>*/IMSConnIVPServletHTMLInput.html

   where *servername* is the name of your Web server machine.
2. Executing the IVP causes the IMS **/STA OTMA** command to be sent to the target IMS machine. Therefore, before clicking **Submit**, be sure to provide the following values for the fields described below.

Figure 5. IMS Connector for Java IVP Input Form



Host name
The TCP/IP host name of the machine running IMS Connect.

Port The port assigned to IMS Connect.

Datastore
The IMS XCF member name. It is also the Datastore ID you specified when you installed IMS Connect.

User ID
The SAF user ID to be passed to OTMA. For example, the RACF user ID.

User ID is optional. However, you can provide it, along with password, if you want to verify your installation's security configuration.

Password
Password is optional (see User ID).

Group name
Group name is optional (see User ID).

When the fields are complete, click **Submit** on the input form to run the Java servlet.

The objective of the IVP is to receive a message from IMS in reply to the **/STA OTMA** command. One of two possible messages is returned, depending on the security configuration of your host environment and whether the given user ID is permitted to issue the **/STA** command. Either message indicates successful execution of the IVP.

- DFS1292E SECURITY VIOLATION
- DFS058I 17:33:32 START COMMAND COMPLETED

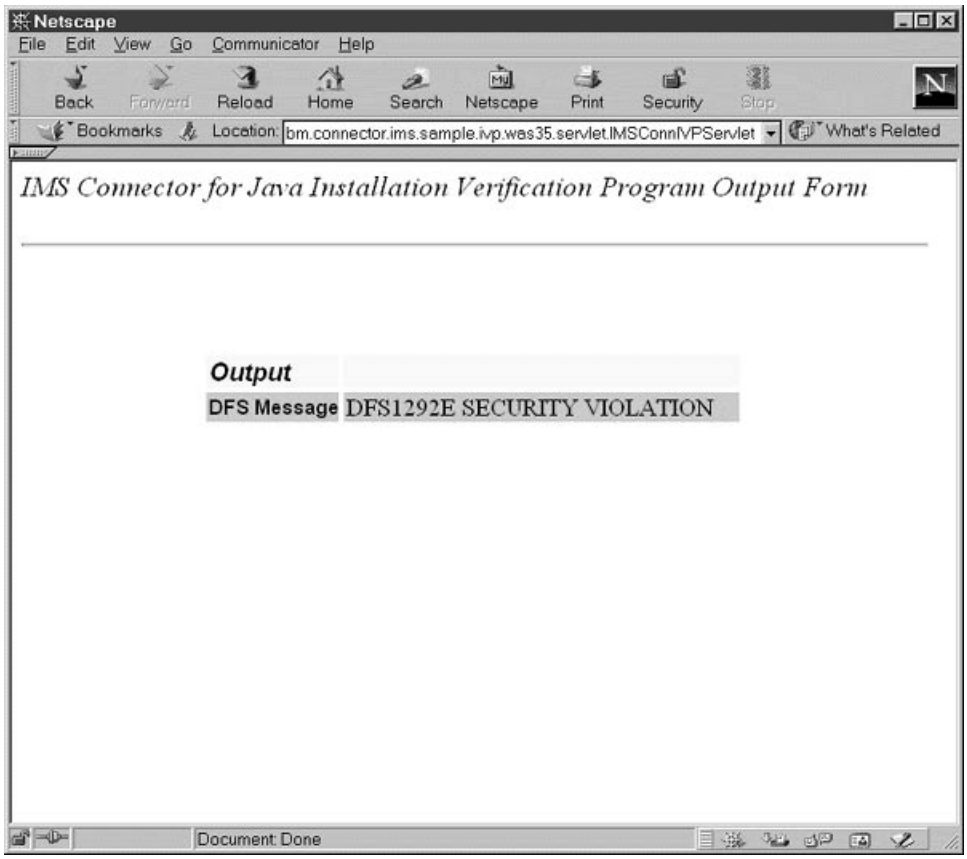

Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: bm.connector.ims.sample.ivp.was35.servlet.IMSConnIVPServlet What's Related
IMS Connector for Java Installation Verification Program Output Form
Output
DFS Message DFS1292E SECURITY VIOLATION
Document: Done

Figure 6. IMS Connector for Java IVP Output Form

This test does not verify that a particular IMS transaction can run successfully; only that the path to IMS is available. If an error occurs, check the debug console or logging file that you declare in Step 3. This file lists exceptions and stack traces that are useful for diagnosing the failure. The following are common causes for IVP failure:

- Host name is misspelled or is not sufficiently qualified.
- Incorrect port.
- TCP/IP failure. Always ensure a successful "ping" prior to running the IVP.
- Datastore name is incorrect or is misspelled. Datastore name must use uppercase characters.
- Wrong level of IMS Connect running on the host system. See Prerequisites for Using IMS Connector for Java for the required level of IMS Connect.

Related Reading: For more information on diagnosing problems with IMS Connector for Java, see Using the trace and error logging facility and Messages and exceptions.

For debugging and tracing information on **WebSphere Application Server**, please see the **Tune and Troubleshoot** section of the WebSphere Application Server documentation.

Chapter 3. Building Java applications and servlets

To access IMS OTMA from a Java application or servlet, IMS OTMA and IMS Connect must be installed and running on the host machine.

The following steps provide a high-level overview of how to build a Java application or servlet that accesses an IMS transaction. The first seven steps can be easily accomplished by building an Enterprise Access Builder (EAB) command using VisualAge for Java and its Visual Composition Editor or Command Editor.

1. Create an IMSConnectionSpec object to be used for information about the connection to IMS Connect on the host machine. Provide the following information to the IMSConnectionSpec object:
   - TCP/IP host name of the machine running IMS Connect
   - Port associated with IMS Connect

   IMS Connector for Java provides an IMSConnectionSpec Java bean for this step.
2. Create an IMSInteractionSpec object to be used for the interaction between the Java application or servlet and IMS OTMA. Provide the following information to the IMSInteractionSpec object:
   - Type of interaction that the Java application or servlet is to have with IMS OTMA (for example, MODE_SEND_RECEIVE and SYNC_LEVEL_NONE)
   - Name of the target datastore. This is the datastore ID that is provided when configuring IMS Connect on the host machine.

   IMS Connector for Java provides an IMSInteractionSpec Java bean for this step.
3. Create an object to represent the input to the IMS transaction. Currently, VisualAge for Java enables you to create a COBOL RecordType, from which an input Java record bean is created. This support uses a COBOL 01 commarea (I/O PCB input/output area descriptor) to define the structure of the input bean.
4. Create an object to represent the output from the IMS transaction. Currently, VisualAge for Java enables you to create a COBOL RecordType, from which an output Java record bean is created. This support uses a COBOL 01 commarea to define the structure of the output bean.

   The next steps are provided by the execute method of the Enterprise Access Builder command that is created in the Command Editor by connecting the Java beans that are created in steps 1-4.
5. Create a Communication object using the createCommunication method of the IMSConnectionSpec object.
6. Connect to the IMS Connect host component by invoking the Communication object's connect method.
7. Execute the request specified in the IMSInteractionSpec object by invoking the execute method of the Communication object.

   After you have an Enterprise Access Builder command that corresponds to an IMS transaction, you can create a Java application to run the transaction by doing the following:
   1. Provide the user ID, password, group name, and a ConnectionManager object to a JavaRuntimeContext object.
   2. Provide the input data for the IMS transaction to the Enterprise Access Builder command by invoking the appropriate set methods.

© Copyright IBM Corp. 1998, 2000

3. Invoke the execute method of the Enterprise Access Builder command to run the IMS transaction.
4. Display the output of the IMS transaction by using the appropriate get methods of the Enterprise Access Builder command to retrieve the output data.

   If you want to build a servlet to run the IMS transaction, do the following:

1. Use the servlet generation wizard in the IBM WebSphere Studio to generate a servlet and associated input and output HTML and JSP files from the Enterprise Access Builder command. The HTML and JSP files are used to provide the input to the IMS transaction and to display the output from the IMS transaction.
2. Run the servlet using the IBM WebSphere Application Server.

Examples of these steps are presented in subsequent sections of this documentation.

Related Reading: For installation and configuration information for IMS Connect, see the IMS Connect documentation at the IMS Web site: http://www.ibm.com/software/data/ims

Building Java Applications and Servlets for Conversational Transactions

The following steps provide a high-level overview of how a Java application or servlet that accesses an IMS
conversational transaction works. You can use this information along with the information in Building Java applications and servlets to build a Java application or servlet that accesses an IMS conversational transaction.

1. The user starts an IMS conversational Web application by displaying the input HTML page of the Web application in a Web browser.
2. The user provides the initial input data of the IMS conversational transaction in the input HTML page. This starts the first iteration of the IMS conversational transaction.
3. The Web browser sends the request with the input data to a Java servlet running in WebSphere Application Server.
4. The Java servlet obtains the HttpSession object and creates an IMSConvContext object and IMSConvUnboundHttpSessionCleanup object. It sets up the IMSConvContext object with the CCF ConnectionManager and saves it in the HttpSession object. The servlet then invokes the execute() method of the EAB command, passing the input data received from the browser. The EAB command uses IMS Connector for Java APIs to send the transaction request to IMS via IMS Connect.
5. IMS receives the request and schedules the IMS conversational application program. The IMS application program processes the request and the reply is returned to the Java servlet.
6. The Java servlet populates the EAB command with the output data of the reply and loads the appropriate result JSP page to display the output data on the Web browser. The Java servlet or the end user to determines if the output data received is acceptable. If it is, the Java servlet invokes the commit() method of JavaCoordinator to accept the output and commit the iteration of the conversation. Otherwise, it invokes the rollback() method of JavaCoordinator to reject the output and back out the iteration of conversation.

7. The user provides the data for the second iteration on the JSP page used to display the output data of the first iteration, then submits the next request to IMS. This invokes the second iteration of the IMS conversational transaction.

Subsequent iterations of the conversation repeat steps 3-7 until the user or the IMS application program terminates the conversation. In step 4, for second and subsequent iterations of the conversation, the Java servlet obtains the IMSConvContext object from the unique HttpSession object.

Note: The steps described above may differ when used with different synchronization levels. The above scenario describes an application flow with SYNC_LEVEL_CONFIRM. When used with SYNC_LEVEL_NONE, Step 6 would not be executed.

Examples of how to create Web applications that perform these similar steps for nonconversational Web applications are presented in other sections of this documentation.

Terminating a Conversation

In the IMS Connector for Java conversational models, an IMS conversation is typically terminated in one of two ways:

- The client terminates the IMS conversational transaction:
  1. The end user submits a request to end the IMS conversation from the browser.
  2. The Java servlet invokes an EAB command with the MODE_DEALLOCATE interaction mode. MODE_DEALLOCATE is a "send only" interaction. IMS terminates the conversation. The transaction is not scheduled and no output message is sent to the servlet from IMS.
  3. The Java servlet should invalidate the HttpSession object and cleanup the connection resource for reuse.
  4. The servlet should also load a customized JSP page to display a message indicating an attempt was made to end the conversation.
- The IMS application program terminates the conversation:
  1. The end user submits a request to IMS to run the IMS conversational transaction. The input message may contain a specific request to terminate the conversation. Please note that the IMS application program may also terminate the conversation based on criteria other than the contents of the input message.
  2. IMS schedules the IMS conversational application program, the program processes the input message, then sends the transaction output message back to the servlet. The OTMA message containing the transaction output message contains a bit signifying that the conversation has been ended. IMS ends the conversation after sending the transaction output message back to the servlet.
  3. IMS Connector for Java, receives the OTMA message and sets the value of the *convEnded* property in the IMSInteractionSpec to true. The EAB command is populated with the output data.
  4. The Java servlet should invalidate the HttpSession object and cleanup the connection resource.
  5. The servlet should also load a customized JSP page to display a message on the Web browser indicating that the conversation was ended.

Chapter 4. Building a Java Application to Run an IMS Transaction

This section describes how to build an Enterprise Access Builder (EAB) command that uses IMS Connector for Java to run an IMS transaction. It includes descriptions of how to create record beans that represent the input and output messages of the IMS transaction, as well as how to use the VisualAge for Java Command Editor to build the EAB command. The section also describes how to create and run a Java application that uses an EAB command.

This example uses the IMS transaction that is described in IMS INSTALL/IVP Sample Application.

This example corresponds to one of the samples included in the IMS Connect or Samples feature. The Java source relating to this example is contained in the com.ibm.connector.ims.sample.phonebook.command package of the Connector IMS Samples project which is added to your workspace when the IMS Connector Samples feature is added. See Preparing your VisualAge for Java Environment for instructions on installing the IMS Connect or Samples feature if you have not already done so. By following the instructions in this section, you will create an application which is the same as the corresponding sample except that it uses different project and package names than are used in the sample since you can't have duplicate package or project names within your VisualAge environment. As a result, you will be able to check your work against the sample, helping you to resolve any difficulties you encounter as you progress through the steps that follow.

Using the Enterprise Access Builder

The Enterprise Access Builder consists of frameworks and tools that allow you to create Java applications that access existing host applications and data. An interaction with a back-end system often involves sending input data to a host application, which would respond with some output data. In the Enterprise Access Builder, this interaction is called an *EAB command*.

EAB commands dictate the data that is passed to and from the back-end system in a single interaction. Records are used as input and output for the EAB commands. Records define the layout of the input and output data within the back-end system. An EAB command is a composite, the main elements being connection specification, interaction specification, input bean, and output bean.

When constructing an EAB command, you must provide the following information:

Input data
The input data is required by the EAB command to perform the execution.

Output data
The output data is the result of the EAB command's execution.

Communication information
This information is a set of classes used to communicate with the host system. A connection specification and an interaction specification define how an EAB command communicates with a host system. You use the ConnectionSpec interface to specify the connection to the host system. You

© Copyright IBM Corp. 1998, 2000

use the InteractionSpec interface to specify to the EAB command which program to call through the connection derived from ConnectionSpec.

The input of an EAB command is defined by a record bean. The output of an EAB command is also defined by a record bean; however, there might exist multiple output record beans defined as candidates. An EAB command can show an entire RecordBean in its interface, or only selected record bean properties.

Using the Command Editor

The Command Editor is a tool that guides you through the steps of constructing a command, allowing you to focus on the specific composition patterns of a command. You can use the Command Editor to create EAB commands from components (beans) representing the input, output, and communication information.

Creating IMS Transaction Input and Output Beans

This section describes how to create IMS transaction input and output beans that you can use in an EAB command that runs an IMS transaction. The steps include the following:

1. Start the Workbench in the VisualAge for Java IDE
2. Create a Project and Package for the Application
3. Create a COBOL RecordType Class for the IMS Transaction Input Message
4. Create a Transaction Input Record Bean
5. Create a COBOL RecordType Class for the IMS Transaction Output Message
6. Create a Transaction Output Record Bean

The COBOL source used in steps 3 and 5 of this example can be found in <*IBM_Connectors_install_dir*>\imsconn\samples\misc\Ex01.ccp, where <*IBM_Connectors_install_dir*> is the IBM Connectors installation directory.

For example, `d:\IBM Connectors`. Initially, <*IBM_Connectors_install_dir*> is **<vaj_install_drive>\IBM Connectors**, where `<vaj_install_drive>` is the drive on which VisualAge for Java is installed.

Step 1: Start the Workbench in the VisualAge for Java IDE

Click the **Start** icon and click **Programs —> IBM VisualAge for Java for Windows —> IBM VisualAge for Java**.

Step 2: Create a Project and Package for the Application

This step creates a project, called `Examples`, that contains a package named `ex01`. Do the following:

1. From the **Selected** menu, click **Add —> Project**. The **AddProject** wizard appears. Enter a name for the new project (for example, `Examples`), and click **Finish**.

2. To add a new package to this project, ensure that the correct project is selected. Then, from the **Selected** menu, click **Add —> Package**. The **AddPackage** wizard appears. Ensure that the correct project name is specified and that the **Create a new package named:** radio button is selected. Enter a name for the new package (for example, `ex01`) and click **Finish**.

Definition:

A *project* is a unit of organization that is used to group packages. A VisualAge for Java project is the starting point for your development work in the IDE.

Step 3: Create a COBOL RecordType Class for the IMS Transaction Input Message

This step parses the COBOL commarea that represents the input message for the IMS transaction, and creates a COBOL RecordType class that encapsulates the information in the commarea. This class will be used in *Step 4: Create a Transaction Input Record Bean* to build a Java bean that represents the input message. Do the following:

1. Ensure that the new package, `ex01`, is selected. From the **Selected** menu, click **Tools —> Enterprise Access Builder —> Import COBOL to RecordType...** The **Import COBOL to RecordType** wizard appears.
2. On the **Import COBOL to RecordType** wizard, do the following:
   - Enter the path and file name of the COBOL file that contains the commarea representing the IMS transaction input message. Click **Next**. The next **Import COBOL to RecordType** wizard appears, and contains a list of the available level 01 commareas.
   - Select the commarea representing the IMS transaction input message from the Available level 01 commarea list (for example, `INPUT_MSG`)) and add it to the **Selected** commareas list by clicking the **>** button.
   - Select the **Use BigDecimal** check box. This will generate the COBOL type as a BigDecimal Java type. Click **Next**.
   - Ensure that the correct names appear in the **Project Name** and **Package** entry fields.
   - Enter a name for the new class that will represent the COBOL RecordType. For example, `InMsgInfo`.
   - To continue to generate the record bean from this record type, ensure that both the **Continue working with newly created record type** check box and the **Create record from record type** radio button are selected.
   - Click **Finish**. A new class named `InMsgInfo` appears in the package named `ex01` in the IDE Workbench.

The following fragment of COBOL source code is the 01 commarea for the IMS transaction input message:

```
01  INPUT-MSG.
           02  IN-LL          PICTURE S9(3) COMP.
           02  IN-ZZ          PICTURE S9(3) COMP.
           02  IN-TRCD        PICTURE X(10).
           02  IN-CMD         PICTURE X(8).
           02  IN-NAME1       PICTURE X(10).
           02  IN-NAME2       PICTURE X(10).
           02  IN-EXTN        PICTURE X(10).
           02  IN-ZIP         PICTURE X(7).
```

Related Reading: For more information on importing COBOL source to a record type, see the *VisualAge for Java Enterprise Access Builder* documentation.

Step 4: Create a Transaction Input Record Bean

This step creates a record bean that represents the transaction input message from the COBOL RecordType class created in *Step 3: Create a COBOL RecordType Class for the IMS Transaction Input Message*. A record bean contains properties that map to the fields in a record in a host program. In this case, the record in the host program is the IMS transaction input message. Do the following:

1. If both the **Continue working with newly created record type** check box and the **Create record from record type** radio button are selected in *Step 3: Create a COBOL RecordType Class for the IMS Transaction Input Message*, the **Create Record from Record Type** wizard appears. To manually start the **Create Record from Record Type** wizard, ensure that the COBOL RecordType class, `InMsgInfo`, is selected. From the **Selected** menu, click **Tools —> Enterprise Access Builder —> Create Record from Record Type... The Create Record from Record Type** wizard appears.
2. On the **Create Record from Record Type** wizard, enter the following:
   - Ensure that the correct names appear in the **Project Name** and **Package** entry fields.
   - Enter a name for the new class that will represent the input message. For example, `InMsg`.
   - Click **Access Method: Direct**.
   - Click **Record Style Custom Records**.
   - In **Additional Options**, ensure that the following check box is selected: **Generate with Notification** Do not check **Use Inner Classes** or **Shorten Names**.
   - Click **Next** to view the next Create Record from Record Type wizard. Ensure that this wizard shows the following values, which indicate that the IMS transaction input message is processed on an MVS host machine:
     – Floating Point Format is `IBM`.
     – Endian is `Big Endian`.
     – Remote Integer Endian is `Big Endian`.
     – Code Page is `037`.
     – Machine Type is `MVS`.
3. Click **Finish** on the Create Record from Record Type wizard. Two new classes, called `InMsgBeanInfo` and `InMsg`, appear in the package `ex01` in the IDE Workbench. The `InMsg` Java bean represents the IMS transaction input message.

Notes:

Code page refers to the data of the IMS transaction. If your transaction data is other than U.S. English (code page 037), you must enter a different code page in this field.

Ensure that all values on the **Create Record from Record Type** wizard have been accepted. That is, all of the field values should appear gray before you select **Finish**.

Related Reading: For more information on additional options, see the *VisualAge for Java Enterprise Access Builder* documentation.

Step 5: Create a COBOL RecordType Class for the IMS Transaction Output Message

Repeat the instructions in *Step 3: Create a COBOL RecordType Class for the IMS Transaction Input Message* to create a class named `OutMsgInfo` from the 01 commarea representing the IMS transaction output message.

The following fragment of COBOL source code is the 01 commarea for the IMS transaction output message:

```
01  OUTPUT-MSG.
           02  OUT-LL       PICTURE S9(3) COMP VALUE +0.
           02  OUT-ZZ       PICTURE S9(3) COMP VALUE +0.
           02  OUT-MSG      PICTURE X(40) VALUE SPACES.
           02  OUT-CMD      PICTURE X(8) VALUE SPACES.
           02  OUT-NAME1    PICTURE X(10) VALUE SPACES.
           02  OUT-NAME2    PICTURE X(10) VALUE SPACES.
           02  OUT-EXTN     PICTURE X(10) VALUE SPACES.
           02  OUT-ZIP      PICTURE X(7) VALUE SPACES.
           02  OUT-SEGNO    PICTURE X(4) VALUE SPACES.
```

Step 6: Create a Transaction Output Record Bean

Repeat the instructions in *Step 4: Create a Transaction Input Record Bean* above to create classes named `OutMsgBeanInfo` and `OutMsg`. The `OutMsg` Java bean represents the IMS transaction output message.

Creating an EAB Command Using the Command Editor

This section describes how you can create an Enterprise Access Builder (EAB) command using the VisualAge for Java Command Editor. The Command Editor guides you through the steps of constructing a command, allowing you to focus on the specific composition patterns of a command.

The EAB command is a composite bean consisting of a ConnectionSpec bean, an InteractionSpec bean, a bean representing the input of the transaction, a bean representing the output of the transaction, and the DFSMsg bean. The DFSMsg bean is used to process messages from IMS other than the output of the transaction, such as error or status messages.

The Java source for this example can be found in the package com.ibm.connector.ims.sample.phonebook.command in the WorkBench. Initially, `<IBM_Connectors_install_dir>` is **<vaj_install_drive>\IBM Connectors**, where `<vaj_install_drive>` is the drive on which VisualAge for Java is installed.

The steps include the following:

1. Create a Command Class for the IMS Transaction
2. Promote the IMS Transaction Input Properties
3. Promote the IMS Transaction Output Properties
4. Promote the DFS Message Properties

5. Promote the Interaction Specification Properties
6. Promote the Connection Specification Properties
7. Create the Command Bean

Step 1: Create a Command Class for the IMS Transaction

This step creates a class for the EAB command. Do the following:

1. Ensure that the package `ex01` is selected. From the **Selected** menu, click **Tools —> Enterprise Access Builder —> Create Command...** The Create Command wizard appears.
2. On the Create Command wizard, do the following:
   a. Ensure that the correct names appear in the **Project Name** and **Package** entry fields.
   b. Enter a name for the new class that represents the EAB command (for example, Ex01Command).
   c. Ensure that the **Edit when finished** check box is selected to edit the Command class when it is created.
   d. Associate a connection specification with the EAB command. Click the **Browse** button. All available ConnectionSpec classes are shown. Choose `IMSConnectionSpec` and click **OK**.
   e. Click **Edit** to edit the connection specification properties. Select the property name and specify the value as follows:

      Connection timeout
      If you want to provide additional connection properties, see the interface **com.ibm.connector.ConnectionSpecManagementProperties** in package **com.ibm.connector** of the Enterprise Access Builder for a description of this property.

      Host name
      Enter the TCP/IP host name of the machine running IMS Connect that the command (IMS transaction) will be using.

      Maximum connections
      If you want to provide additional connection properties, see the interface **com.ibm.connector.ConnectionSpecManagementProperties** in package **com.ibm.connector** of the Enterprise Access Builder for a description of this property.

      Minimum connections
      If you want to provide additional connection properties, see the interface **com.ibm.connector.ConnectionSpecManagementProperties** in package **com.ibm.connector** of the Enterprise Access Builder for a description of this property.

      Port Enter the port for IMS Connect that the command (IMS transaction) will be using.

      Realm The **Realm** property defines a unique name for a remote target to which the client Java application or servlet is connecting. The **Realm** property is used by the infrastructure LogonInfo of the component server for mapping security information. For example, it

maps the client information to the user information. The component server that supports the **Realm** property provides the appropriate value.

Reap time
If you want to provide additional connection properties, see the interface **com.ibm.connector.ConnectionSpecManagementProperties** in package **com.ibm.connector** of the Enterprise Access Builder for a description of this property.

Unused timeout
If you want to provide additional connection properties, see the interface **com.ibm.connector.ConnectionSpecManagementProperties** in package **com.ibm.connector** of the Enterprise Access Builder for a description of this property.

f. Associate an interaction specification with the EAB command. Click the **Browse** button. All available InteractionSpec classes are shown. Choose `IMSInteractionSpec` and click **OK**.

g. Click **Edit** to edit the interaction specification properties. Select the property name and specify the value as follows:

Datastore name
Enter the name of the target IMS datastore for this command (IMS transaction). The Datastore Name is a maximum of 8 uppercase characters.

LTERM Name
Leave as is.

Map Name
Leave as is.

Mode Leave as MODE_SEND_RECEIVE

Synchronization Level
Leave as SYNC_LEVEL_NONE

h. Click **Next** to use the Add Input/Output Beans wizard.

3. On the Add Input/Output Beans wizard, enter the following:

a. In the Input record bean section, ensure that the **Implements IByteBuffer** check box is selected. Enter the class name of the Transaction Input bean. Click the **Browse** button to display a list of all available classes that implement IByteBuffer. Select the appropriate **Type Name** (`InMsg`) and **Package Name** (`ex01`) and click **OK**.

b. Provide the Mapper class, if necessary. In this example, no additional Mapper class is needed.

c. In the Output record bean section, ensure that **Select output record beans** is selected. Click the **Add** button. A dialog box appears to enter the output record bean. In this dialog box, ensure that the **Implements IByteBuffer** check box is selected. Enter the class name of the Transaction Output bean . Click the **Browse** button to display a list of all available classes that implement IByteBuffer. Select the appropriate **Type Name** (`OutMsg`) and **Package Name** (`ex01`) and click **OK**.

d. Repeat the previous step to add additional output record beans, if necessary. In this example, repeat the previous step to add the output

record bean with DFSMsg as **Type Name** and com.ibm.connector.imstoc as the **Package Name**. Provide the Mapper class, if necessary.

e. You can use **Modify...** or **Delete** to modify or delete the output record beans from the list.

4. Click **Finish** to generate the EAB command. If the **Edit with finished** check box is selected in Item 2, the new EAB command appears in the Command Editor.
5. You can use the Command Editor to create a new command, open existing commands, add or delete beans and properties, and promote or change properties. To launch an existing command with the Command Editor, ensure that the EAB command class is selected, then select **Tools —> Enterprise Access Builder —> Edit Command...**

To change the bean properties with the Command Editor, do the following:

1. Select a folder icon (for example, `Connector`) under the command bean icon (for example, `ex01.Ex01Command`) in the top left pane. The bean appears in the top right pane.
2. Select the bean (`com.ibm.connector.imstoc.IMSConnectionSpec`) in the top right pane. The properties and their values appear in the bottom pane.
3. Select the property name that you want to change and specify the new value.

Related Reading: For more information on the Mapper class, see the *VisualAge for Java Enterprise Access Builder* documentation.

Step 2: Promote the IMS Transaction Input Properties

When creating a composition bean (such as the EAB command bean) you might want some features of beans that are embedded within it to appear in the interface of the composite bean. To add features of a transaction input bean to the interface of a composite bean, promote them to the composite's interface by doing the following:

1. In the Command Editor, select the **Input** folder under the `ex01.Ex01Command` icon in the top left pane. The input bean appears in the top right pane.
2. Select the input bean (`ex01.InMsg`). The properties and their values appear in the bottom pane.
3. Select the **IN__LL** property in the bottom pane with the right mouse button. Choose **Promote Property** on the pop-up context menu. The promoted property is marked with a green dot in front of the property name. By default, properties are promoted with the same name as the originating bean.
4. Repeat the previous step to promote the following properties:
   - IN__ZZ
   - IN__TRCD
   - IN__CMD
   - IN__NAME1
   - IN__NAME2
   - IN__EXTN
   - IN__ZIP

Step 3: Promote the IMS Transaction Output Properties

To add features of an output transaction bean to the interface of a composite bean, promote them to the composite's interface by doing the following:

1. In the Command Editor, select the **Output** folder under the `ex01.Ex01Command` icon in the top left pane. The output beans appear in the top right pane.
2. Select the output bean (`ex01.OutMsg`). The properties and their values appear in the bottom pane.
3. Select the **OUT__LL** property in the bottom pane with the right mouse button. Choose **Promote Property** on the pop-up context menu. The promoted property is marked with a green dot in front of the property name. By default, properties are promoted with the same name as the originating bean.
4. Repeat the previous step to promote the following properties:
   - OUT__ZZ
   - OUT__TRCD
   - OUT__CMD
   - OUT__NAME1
   - OUT__NAME2
   - OUT__EXTN
   - OUT__ZIP

Step 4: Promote the DFS Message Properties

To add features of embedded beans to the interface of a composite bean, you must promote them to the composite bean's interface. In the case of the DFSMsg message bean, two options exist for promoting properties:

- Option 1

  If you want to use the EAB command in a Java application program (rather than in a Java servlet), promote either the getDFSMessage() method or the getDFSMessageSegments() method. The method getDFSMessage returns the message as a string of its concatenated segments. Method getDFSMessageSegments() returns each segment of the message as a string.

  Since the Command Editor does not allow promotion of methods, you must add the appropriate method to the EAB Command as follows:

  1. Right-click on the EAB Command (for example, Ex01Command).
  2. Select **Add—>Method...** The ″Create Method″ SmartGuide appears.
  3. Select the **Create a new method** radio button on the ″Create Method″ SmartGuide, then press **Next.** The ″Attributes″ SmartGuide appears.
  4. If the method being added is getDFSMessage():
     a. Enter getDFSMessage in the **Method Name:** entry field.
     b. Enter String in the **Return Type:** entry field.
     c. Select the **public** radio button in **Access Modifiers**, then press **Finish**.
     d. Modify the code of the newly added method by replacing the statement ″return null;″ with the following statement:

        ```
        return getceOutput1().getDFSMessage();
        ```

        where getceOutput1() is the method that returns the corresponding message bean.

        For example, replace the statement ″return null;″ with

```
return getceOutput1().getDFSMessage();
```

If the method being added is getDFSMessageSegments():

1. Enter getDFSMessageSegments in the **Method Name:** entry field.
2. Enter java.util.Vector in the **Return Type:** entry field.
3. Select the **public** radio button in Access Modifiers, then press **Finish**.
4. Modify the code of the newly added method by replacing the statement ″return null;″ with the following statement:

   ```
   return getceOutputi().getDFSMessageSegments()
   ```

   where getceOutputi() is the method that returns the corresponding message bean. For example, replace the statement ″return null;″ with

   ```
   return getceOutput1().getDFSMessageSegments();
   ```

- Option 2

  If you want to use the EAB command to generate a Java servlet using WebSphere Studio, promote one or more data properties of the message, rather than the getDFSMessage() method or the getDFSMessageSegments() method. The WebSphere Servlet Generation wizard currently only supports string properties for display on the output HTML.

  Note:

  Methods getDFSMessage() and getDFSMessageSegments() can still be used in .jsp pages, but must be manually coded rather than generated by WebSphere Studio.

  To promote data properties of the message:

  1. In the Command Editor, select the Output folder under the ex01.Ex01Command icon in the top left pane. The output beans appear in the top right pane.
  2. Select the DFS message bean (com.ibm.connector.imstoc.DFSMsg). The properties and their values appear in the bottom pane.
  3. Select, for example, the DFSDATA1 property in the bottom pane with the right mouse button. Choose **Promote Property** on the pop-up context menu. The promoted property is marked with a green dot in front of the property name. By default, properties are promoted with the same name as the originating bean.

Step 5: Promote the Interaction Specification Properties

If you want your Java application or servlet to accept values for IMSInteractionSpec properties as input, you must promote features of the embedded bean to the interface of the EAB Command bean. For example, to provide input values for Datastore Name, do the following:

1. In the Command Editor, select the **Connector** folder under the `ex01.Ex01Command` icon in the top left pane. The connector beans appear in the top right pane.
2. Select the IMS interaction specification bean (com.ibm.connector.imstoc.IMSInteractionSpec). The properties and their values appear in the bottom pane.
3. Select the **Datastore name** property in the bottom pane with the right mouse button. Choose **Promote Property** on the pop-up context menu. The promoted

property is marked with a green dot in front of the property name. By default, properties are promoted with the same name as the originating bean.

Step 6: Promote the Connection Specification Properties

If you want your Java application or servlet to accept values for IMSConnectionSpec properties as input, you must promote features of the embedded bean to the interface of the EAB Command bean. For example, to provide input values for Hostname or Port, do the following:

1. In the Command Editor, select the **Connector** folder under the `ex01.Ex01Command` icon in the top left pane. The connector beans appear in the top right pane.
2. Select the IMS connection specification bean (com.ibm.connector.imstoc.IMSConnectionSpec). The properties and their values appear in the bottom pane.
3. Select the **Host name** property in the bottom pane with the right mouse button. Choose **Promote Property** on the pop-up context menu. The promoted property is marked with a green dot in front of the property name. By default, properties are promoted with the same name as the originating bean.
4. Repeat the previous step to promote the **Port** property.

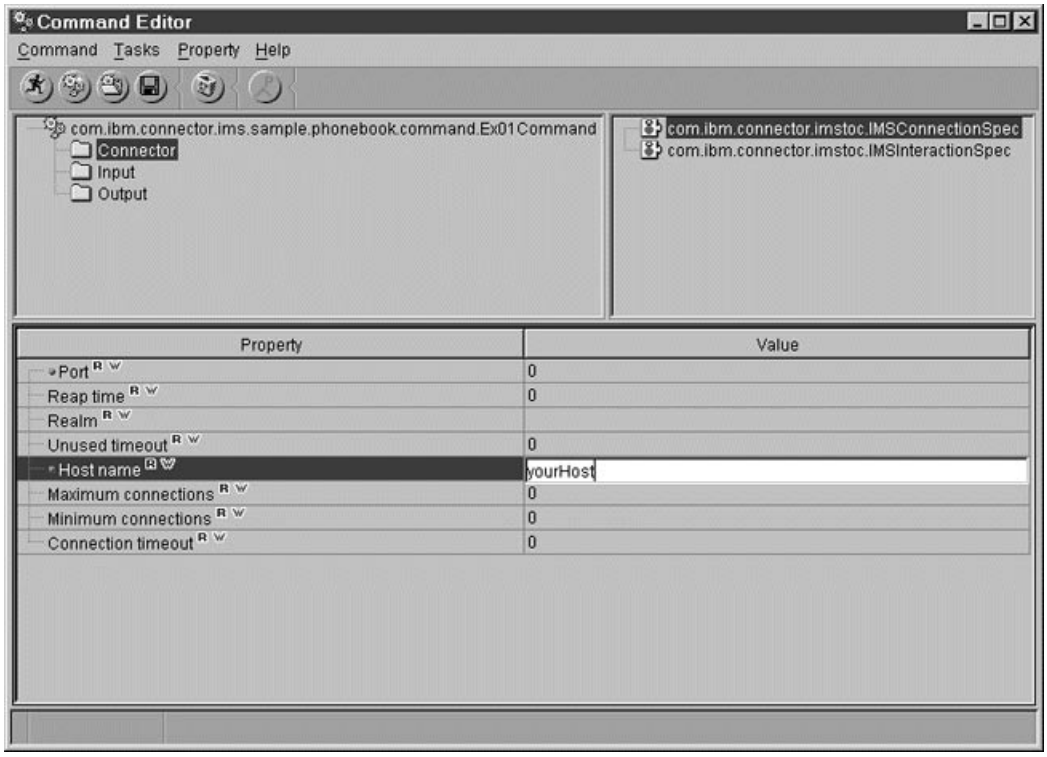


Figure 7. The Completed Command Bean as shown in the Command Editor

Step 7: Create the Command Bean

To save the command bean, select **Command —> Save** from the menu bar. Select **Command —> Exit Editor** from the menu bar to exit the Command Editor. The Workbench now shows two classes associated with the EAB command: `Ex01Command` and `Ex01CommandBeanInfo`.

An EAB command that is created using the Create Command wizard and the Command Editor can only be opened and edited using the Command Editor. An EAB command that is created using the Visual Composition Editor must be edited using the Visual Composition Editor. When using the Command Editor to open an EAB command that is created using the Visual Composition Editor, you are prompted to convert the command to the appropriate format for the Command Editor. After this conversion, however, it must be edited with the Command Editor.

Writing a Java Application to Use the EAB Command

This section demonstrates how you can use the EAB command that was created in *Creating an EAB Command Using the Command Editor* to create a Java application. You can use the Java application to run the IMS transaction. This example illustrates only basic concepts, and does not include sophisticated input and output processing.

The steps include the following:

1. Start the Workbench in the VisualAge for Java IDE
2. Create an Executable Class
3. Add Java Code to the Class

Java source for the EAB command that you created in *Creating an EAB Command Using the Command Editor* can be found in the Package com.ibm.connector.ims.sample.phonebook.command in the Workbench.

Step 1: Start the Workbench in the VisualAge for Java IDE

If you are using VisualAge for Java on Windows, click the **Start** icon and click **Programs —> IBM VisualAge for Java for Windows —> IBM VisualAge for Java**. If the **Welcome to VisualAge** dialog box appears, click **Go to the Workbench** and click **OK**. The IDE workbench appears.

Step 2: Create an Executable Class

1. From the **Selected** menu, click **Add —> Class**. A wizard appears to request all of the necessary information to create a class. Do the following:
   a. Ensure that the **Project** field contains your project name, `Examples`.
   b. Ensure that the **Package** field contains your package name, `ex01`.
   c. Click **Create a new class**.
   d. Enter `Ex01Execute` in the **Class name** field.
   e. Ensure that the **Superclass** is `java.lang.Object`.
   f. Ensure that the **Compose the class visually** check box is *not* checked.
   g. Click **Next** to continue to the next window.
2. You need to add six import statements to a class that executes an EAB command that uses IMS Connector for Java. To include these classes as import statements, click **Add Package...**. A list of available packages appears. From the list, select each of the following packages, and click **Add** to include it in the import statements.

- com.ibm.connector.connectionmanager
- com.ibm.connector.imstoc
- com.ibm.connector.infrastructre
- com.ibm.connector.infrastructure.java

After adding the final one, click **Close**.

In addition, this example requires the following import statements:

```
import java.util.StringTokenizer;
```

3.

Ensure that only the following fields are checked:

- public (in modifiers section)
- Methods which must be implemented (Recommended)
- Copy constructors from superclass (Recommended)
- main(String[])

4. To generate the class, click **Finish**. The class appears inside the package that you have specified.

Step 3: Add Java Code to the Class

Add the code below to the main(String[]) method of the Ex01Execute class that you previously created. Replace the values `yourUID`, `yourPwd`, and `yourGrp` with the user ID, password, and group name for your IMS environment.

In addition, the code below illustrates three ways in which error or status messages from IMS can be printed. In each case, the appropriate method or property must be promoted to the EAB command interface.

```
public static void main(String args[])
{
  //-----------------------------------+
  // Process input arguments.                                          |
  // Each argument will be in the format of: Key=Value                 |
  //                                                                   |
  // The arguments will be processed to get the value for the          |
  // following Keys:                                                   |
  //   HOST NAME  - the TCP/IP host name of the machine running IMS    |
  //                TCP/IP OTMA Connection (ITOC) that the command     |
  //                (IMS transaction) will be using.                   |
  //   PORT      - port that IMS Connect          |
  //                (ITOC) will be using on the host machine.          |
  //   DATASTORE - the name of the target IMS datastore that the       |
  //                command (IMS transaction) will be using.           |
  //                                                                   |
  // For example, an input argument might be:                          |
  // argument should be in the format:                                 |
  //   HOSTNAME=HOST1 PORT=1234 DATASTORE=MYIMS                        |
  //-----------------------------------+
  int numArgs = 0;
  String hostName  = null; // Use value in IMSConnectionSpec.
  int  port = 0; // Use value in IMSConnectionSpec.
  String dataStore = null; // Use value in IMSInteractionSpec.

  java.util.Enumeration en = null;

  numArgs = args.length;
  if( numArgs > 0 )
  for( int i = 0; i < numArgs; i++ )
  {
```

```
      StringTokenizer st = new StringTokenizer( args[i], "=" );
      String key = st.nextToken();
      String val = st.nextToken();
      if( key.equalsIgnoreCase( "HOSTNAME" ) )
      {
        if( val != null )
          hostName = new String( val );
      }
      if( key.equalsIgnoreCase( "PORT" ) )
      {
        if( val != null )
          port = java.lang.Integer.parseInt( val, 10 );
      }
      if( key.equalsIgnoreCase( "DATASTORE" ) )
      {
        if( val != null )
          dataStore = new String( val );
      }
    }
    //-------------------------------------+
    // Establish current runtime context.                              |
    //-------------------------------------+
    JavaRuntimeContext runtimeContext = new JavaRuntimeContext();
    JavaRuntimeContext.setCurrent(runtimeContext);

    //-------------------------------------+
    // Set trace level and get a connection manager.                   |
    //-------------------------------------+
    JavaRASService aRASService =
      (JavaRASService)runtimeContext.getRASService();
    aRASService.setTraceLevel(RASService.RAS_TRACE_OFF);
  // aRASService.setTraceLevel(RASService.RAS_TRACE_ENTRY_EXIT);
  // aRASService.setTraceLevel(RASService.RAS_TRACE_INTERNAL);
    ConnectionManager aConnMgr =
      new ConnectionManager();
    runtimeContext.setConnectionManager(aConnMgr);
    try
    {
      ex01.Ex01Command cmd =
        (Ex01Command)java.beans.Beans.instantiate( null, "ex01.Ex01Command");
      if( hostName != null )
      {
        cmd.setHostName( hostName );
      }
      if( port != 0 )
      {
        cmd.setPortNumber( port );
      }
      if( dataStore != null )
      {
        cmd.setDataStoreName( dataStore);
      }
      //-------------------------------------+
      // Replace values yourUID, yourPwd, and yourGrp in the code    |
      // below with values appropriate to your environment.          |
      //-------------------------------------+
      IMSLogonInfoItems logon =
        new IMSLogonInfoItems( runtimeContext.getLogonInfo(),
                               cmd.getConnectionSpec() );
      logon.setUser("yourUID");
      logon.setPassword("yourPwd");
      logon.setGroup("yourGrp");
      //-------------------------------------+
      // Populate the command bean with input data for the           |
      // IMS transaction.                                            |
      //-------------------------------------+
      cmd.setIN__LL( (short)((InMsg)cmd.getInput()).getSize());
```

```
cmd.setIN__ZZ( (short)0 );
cmd.setIN__TRCD(  "IVTNO    ");
cmd.setIN__CMD("DISPLAY");
cmd.setIN__NAME1("Last1");
cmd.setIN__NAME2( "" );
cmd.setIN__EXTN(  "" );
cmd.setIN__ZIP(   "" );
//-------------------------------+
// Invoke the execute method on the command in order to run    |
// the IMS transaction.                                        |
//-------------------------------+
cmd.execute();
//-------------------------------+
// Get the command's output object (bean) and determine its    |
// type.                                                       |
//-------------------------------+
Object bean = cmd.getOutput();
if( java.beans.Beans.isInstanceOf( bean, OutMsg.class ) == true )
{
  //-------------------------------+
  // Retrieve the output of the IMS transaction from the      |
  // command object and print it.                            |
  //-------------------------------+
  System.out.println( "\nOuput from IVTNO is..." +
                      "\nLL: " + cmd.getOUT__LL() +
                      "\nZZ: " + cmd.getOUT__ZZ() +
                      "\nMSG: " + cmd.getOUT__MSG()
                      "\nCMD: " + cmd.getOUT__CMD() +
                      "\nNAME1: " + cmd.getOUT__NAME1() +
                      \nNAME2: " + cmd.getOUT__NAME2() +
                      "\nEXTN: " + cmd.getOUT__EXTN() +
                      "\nZIP: " + cmd.getOUT__ZIP() +
                      "\nSEGNO: " + cmd.getOUT__SEGNO() );
}
else
{
  //-------------------------------+
  // Print message from IMS.                                 |
  //-------------------------------+
  System.out.println( "\nError message using DFSMsg property DFSDATA1 is:"
                      + "\n" + cmd.getDFSDATA1() );
  // Print DFS message as single string consisting of all segments.
  System.out.println( "\nError message using DFSMsg method getDFSMessage is:"
                      + "\n" + cmd.getDFSMessage() );
  // Print individual segments of DFS message.
  en = (cmd.getDFSMessageSegments()).elements();
  System.out.println( "\nError message using DFSMsg method getDFSMessageSegments is:" );
  while( en.hasMoreElements() )
  {
    System.out.println( en.nextElement() );
  }
}
//-------------------------------+
// Commit resources.                                           |
//-------------------------------+
((JavaCoordinator)runtimeContext.getCoordinator()).commit();
}
catch ( Exception e )
{
  System.out.println( "\nCaught exception is: " + e );
}
  //-------------------------------+
  // Close the runtime context.                               |
```

```
        //-------------------------------+
        runtimeContext.close();
        runtimeContext.removeCurrent();
}
```

This sample code allows you to change to a different host name, port, or datastore name at run time by supplying new values as command-line arguments. The format of these arguments is illustrated in the comments of this code.

Testing Your Application within VisualAge for Java

After you have successfully compiled your Java application (for example, `Ex01Execute`), you can test the application from within VisualAge for Java. Do the following:

1. Select the class of your application and, from the **Selected** menu, click **Properties**. The **Properties** window (for example, `Ex01Execute`) appears.
2. Ensure that the **Program** tab is selected and, if required, enter any command-line arguments in the text box under **Command line arguments**. Each argument is separated with a space. For example, in *Writing a Java Application to Use the EAB Command*, you enter:

   ```
   HOSTNAME=yourhost PORT=yourportnumber DATASTORE=yourdatastorename
   ```
3. Select the **Class Path** tab and make sure that the **Include '.' (dot) in class path** and **Project path:** check boxes are checked.
4. Click **Compute Now**, and click **OK** when the class path is computed. This updates the VisualAge for Java class path to include your Java application.
5. Select the main method of your application and from the **Selected** menu, click **Run —> Run main....** The results of running your application appear in the **Console** window.

Running Your Application Outside of VisualAge for Java

If you want to run your Java application on a workstation that does not have VisualAge for Java installed, you must provide access on the workstation to all of the class libraries that the application uses. These class libraries are the same as those that are normally part of VisualAge for Java. Do the following:

1. Ensure that JDK or JRE Version 1.2.2 or higher is installed.
2. Copy the following class libraries (.zip files) to your workstation:

   IBM Common Connector Framework
   File `ccf.zip` is located in the directory
   <IBM_Connectors_install_dir>\classes

   IBM Enterprise Access Builder Library
   File `eablib.zip` is located in the directory
   <install_dir>\eab\runtime30.

   IBM Java Record Library
   File `recjava.zip` is located in the directory
   <IBM_Connectors_install_dir> \eab\runtime30.

   IMS Connector for Java
   File imsconn.zip is located in the directory
   <vaj_intallation_drive>\<*IBM_Connectors_install_dir*>\classes
   where <IBM_Connectors_install_dir> is the IBM Connectors

installation directory, and <vaj_installation_drive> is the VisualAge for Java installation drive (for example, d:\IBMVJava).

In addition, you must create a `.jar` file from the `.class` files for your application and copy it to your workstation.

Your Java application might consist of one or more EAB command beans. Therefore, when you create the `.jar` file for your application, you must include all of the Java beans of which the composite EAB command beans are comprised, as well as any classes that represent the application.

3. Modify (or append to) your workstation's CLASSPATH environment variable to point to the above `.zip` and `.jar` files.
4. Use **java.exe** to run the class that corresponds to your Java application. If your application requires input command-line arguments, enter:

   ```
   java application_class_name arg1 arg2 ...
   ```

   where `application_class_name` is the name of the executable class corresponding to your application and `arg1` and `arg2` are the command-line arguments. For example, java `ex01.Ex01Execute HOSTNAME=MYHOST PORT=9999 DATASTORE=IMSA`.

Note: Consider the following if you are having trouble running your application outside of VisualAge for Java:

- Ensure that all class libraries (.jar and .zip files) are pointed to correctly by the `CLASSPATH` environment variable. A small error in `CLASSPATH` can result in a class not being found. Remember that a GUI application, such as the IMS Connector for Java's IVP, uses classes that may not be part of the JDK or JRE you are using. For example, you may have to separately deploy the ″swing″ classes and point to them with `CLASSPATH`.
- Remember that, in some environments, you may have to restart the process in which you are running your application, after modifying `CLASSPATH`.
- Ensure that you correctly specify the name of your application. Keep in mind case sensitivity and remember to qualify the name of your application by its package name, if applicable.
- Verify that the .jar file for your application contains all of the necessary classes.

Chapter 5. Building a Java Application to Run a Navigator

This section describes how you can create an EAB Navigator to run multiple IMS transactions. An EAB Navigator is composed of multiple EAB commands that form a more complex interaction (navigation) with a host system. This section also describes how to create and run a Java application that uses an EAB Navigator.

The process of creating and running a Java application to run two IMS transactions is described in the following sections:

- Creating a Navigator Using the Command and Visual Composition Editors
- Writing a Java Application to Use a Navigator
- Testing the Java Application

Creating a Navigator Using the Command and Visual Composition Editors

This example illustrates how you can create an EAB Navigator command to run multiple IMS transactions. The Navigator command is composed of other commands that form a more complex interaction (navigation) with a host system. You construct the commands contained in the Navigator using the Command Editor and then create the complex interactions in the Navigator by interconnecting these commands using the Navigator's Visual Composition Editor.

In this example, we build a Navigator command that runs two IMS transactions sequentially. This Navigator command consists of two commands. Each command corresponds to an IMS transaction. The Navigator command accepts input from the user which it uses as the input to the first command. The first command executes and the output from this command is displayed by the Navigator on its output screen and used as the input for the second command. The Navigator then executes the second command and uses the output of this command to formulate the output which it displays on its output screen.

Definition:

An *Enterprise Access Builder Navigator* wraps multiple interactions with a host system into what appears to the user to be a single interaction. From the outside, a Navigator looks like an EAB command. A Navigator consists of both EAB commands and Navigators strung together to form a more complex interaction with the host system.

The following procedure assumes that you have the com.ibm.connector.ims.sample.phonebook.command package on your Workbench. This will be the case if you have installed the IMS Connector Samples feature (and have not subsequently deleted the package.)

The IMS transactions executed by both of the commands that make up the Navigator command take a single-segment input message and return a single-segment output message.

The steps include the following:

1. Start the Workbench in the VisualAge for Java IDE
2. Create a Project and Package for the Application
3. Create a Navigator Bean for Building Complex Commands
4. Set Up the Property Features

© Copyright IBM Corp. 1998, 2000

5. Create a Command Bean for the First Command
6. Create a Command Bean for the Second Command
7. Set Up the Connection Specification for the Navigator (Optional)
8. Promote Connection and InteractionSpec Property Features (Optional)
9. Save the Navigator Bean

Step 1: Start the Workbench in the VisualAge for Java IDE

For a description of how to start the Workbench in the VisualAge for Java IDE, see Step 1: Start the Workbench in the VisualAge for Java IDE (page 36) under Creating IMS Transaction Input and Output Beans (page 36) in Building a Java Application to Run an IMS Transaction.

Step 2: Create a Project and Package for the Application

This step creates a project, called `Examples`, that contains a package named `ex02`.

For a description of how to create a project and package, see Step 2: Create a Project and Package for the Application (page 36) under Creating IMS Transaction Input and Output Beans (page 36) in Building a Java Application to Run an IMS Transaction. The package used by the sample code in this section is `ex02`.

Step 3: Create a Navigator Bean for Building Complex Commands

This step creates the Navigator bean, which is composed of two command beans that correspond to the two IMS transactions. Do the following:

1. With the package `ex02` selected, click the **Selected** menu, then click **Add —> Class...** The Create Class wizard appears.
2. Ensure that the correct names, Examples and ex02, appear in the **Project** and **Package** entry fields, respectively.
3. Click **Create a new class** and enter a name for the new Navigator command class (for example, `Ex02Navigator`).
4. To choose the superclass for class `Ex02Navigator`, click **Browse**, then select **CommunicationNavigator** from the **TypeNames** list.

Ensure that the **Compose the class visually** radio button is selected and click **Finish**. A new window (titled Ex02Navigator [mm/dd/yy HH:MM:SS AM/PM] in Ex02) appears with the **Visual Composition** tab selected.

When you execute a Navigator, the Navigator provides its input to the commands and Navigators of which it is composed in accordance with the connections that are set up when the Navigator is created. These connections in a Navigator also determine the order of execution of each EAB command or Navigator contained in the Navigator.

For a simple Navigator, after the final interaction is carried out, the output of the individual commands and Navigators can be used to formulate the output of the Navigator. For more complex Navigators, the output of the Navigator is determined by the output of the individual commands and Navigators it contains as well as code that can be added to the connections in the Navigator and/or code in the Java application that executes the Navigator.

Step 4: Set Up the Property Features

This step creates new property features for the Navigator command. These property features are linked to the property features of the individual commands that make up the Navigator, which allows the input data and output results to be passed between the individual commands and the Navigator in accordance with the property-to-property connections set up in the Navigator. This enables the output of one command to be used for the input of a subsequent command and allows the output of a command to be passed to the Navigator where it can be processed if necessary and then in turn, passed on to the application executing the Navigator where it can be further processed if necessary and displayed to the user.

Create new property features for the data beans in the command bean by doing the following:

1. Create Input property features:
   - Input for first command:
     - Select the **BeanInfo** tab of the `Ex02Navigator` bean window (located at the top of the window that opened in the last step).
     - Click **Features —> New Property Feature...** from the pull-down menu at the top of the screen. The New Property Feature SmartGuide appears.
     - In the New Property Feature SmartGuide, enter a name in the **Property name** field for one of the promoted properties in the `Ex02FirstCommand`. For example, enter `FIRST_CMD_IN_LL` for the `IN_LL` property of `Ex02FirstCommand`.
     - In the **Property Type** field, select the corresponding type for the property (for example, `short`).
     - Accept the default settings for all of the other selections for this property feature and click **Finish**. A new property feature of `FIRST_CMD_IN_LL` appears in the **Feature** section of the window.
     - Repeat the above steps to create the following new features for all of the other input properties of the `Ex02FirstCommand` bean:

       The following is **Property Type** short:

       - FIRST_CMD_IN_ZZ (for IN_ZZ property of `Ex02FirstCommand`)

       The following are **Property Type** java.lang.string:

       - FIRST_CMD_IN_TRCD (for IN_TRCD property of `Ex02FirstCommand`)
       - FIRST_CMD_IN_CMD (for IN_CMD property of `Ex02FirstCommand`)
       - IN_NAME1 (for IN_NAME1 property of `Ex02FirstCommand`)
       - IN_NAME2 (for IN_NAME2 property of `Ex02FirstCommand`)
       - IN_EXTN (for IN_EXTN property of `Ex02FirstCommand`)
       - IN_ZIP (for IN_ZIP property of `Ex02FirstCommand`)
   - Input for second command:

     Similarly, create the following new features for all of the input properties of the `Ex02SecondCommand` bean.

     The following are **Property Type** short:

     – SECOND_CMD_IN_LL (for IN_LL property of `Ex02SecondCommand`)

     – SECOND_CMD_IN_ZZ (for IN_ZZ property of `Ex02SecondCommand`)

     The following are **Property Type** java.lang.string:

- SECOND_CMD_IN_TRCD (for IN_TRCD property of Ex02SecondCommand)
- SECOND_CMD_IN_CMD (for IN_CMD property of Ex02SecondCommand)

2. Create output property features:

   This step creates the following new features for all of the output properties.

   The following are **Property Type** short:

   - OUT_LL (for OUT_LL property of Ex02SecondCommand bean)
   - OUT_ZZ (for OUT_ZZ property of Ex02SecondCommand bean)

   The following are **Property Type** java.lang.string:

   - OUT_MSG (for OUT_MSG property of Ex02SecondCommand bean)
   - OUT_CMD (for OUT_CMD property of Ex02SecondCommand bean)
   - OUT_NAME1 (for OUT_NAME1 property of Ex02SecondCommand bean)
   - OUT_NAME2 (for OUT_NAME2 property of Ex02SecondCommand bean)
   - OUT_EXTN (for OUT_EXTN property of Ex02SecondCommand bean)
   - OUT_ZIP (for OUT_ZIP property of Ex02SecondCommand bean)
   - OUT_SEGNO (for OUT_SEGNO property of Ex02SecondCommand bean)

Step 5: Create a Command Bean for the First Command

This step creates the first command bean, which represents the first IMS transaction for this example. There are multiple ways in which this command bean can be created. In this example we will "create" the Ex01FirstCommand EAB command bean by reusing the command bean that was created in the ex01 example, Building a Java Application to Run an IMS Transaction, under Creating an EAB Command Using the Command Editor (page 39).

1. Select the **Visual Composition** tab of the Ex02Navigator bean window to start the Visual Composition Editor.
2. Click the **Choose Bean...** icon in the tool palette and ensure that the **Class Bean Type** is selected. Click the **Browse** button for the **Class name** field. In the **Pattern** field, enter Ex01. A list of class names appears. Select Ex01Command for the **Class name** from the list of available classes and com.ibm.connector.ims.sample.phonebook.command from the package name. Click **OK**. Enter Ex02FirstCommand in the Name field and click **OK**.
3. To plant the bean, move the cursor to a position in the frame window and click the left mouse button.
4. The following connections are needed for communication between the command and the Navigator:

   a. UnsuccessfulExecution event connection:

      These steps create an Event-to-method connection such that the returnExecutionUnsuccessful method of the Navigator is invoked when an executionUnsuccessful of the Ex02FirstCommand event occurs.

      - Select the **Ex02FirstCommand** icon and click the right mouse button. Choose **Connect —> Connectable Features...** in the resulting pop-up window.
      - Click **Event** and scroll down the list to choose the executionUnsuccessful(com.ibm.ivj.eab.command.CommandEvent) event and click **OK**.
      - Move the cursor to an empty area in the frame window and click the left mouse button. Click **Connectable Features...**. A list in an **End connection to [Ex02Navigator]** dialog box appears.

- In the **End connection** dialog box, ensure that the **Method** radio button is selected. Choose `returnExecutionUnsuccessful(com.ibm.ivj.eab.command.CommandEvent)` from the scroll-down list and click **OK**. A new dotted-line connection is created.
- Select the new connection and click the right mouse button. Select **Properties** from the pop-up menu, and the **Event-to-method connection properties** window appears. Click the **Pass** event data box and then click **OK**.

The connection now becomes a solid line.

b. StartExecution event connection:

See the description in item **4a** of this step to create an Event-to-method connection between the `internalExecutionStarting(com.ibm.ivj.eab.command.CommandEvent)` event of the Navigator and the `execute(com.ibm.ivj.eab.command.CommandEvent)` of `Ex02FirstCommand`. The execute method of `Ex02FirstCommand` is invoked when an internalExecutionStarting of the Navigator method occurs. Note that this step requires you to start the connection by right-clicking on an open area in the Navigator Visual Composition window and then end the connection by clicking on the Ex02FirstCommand icon. Be sure to click the **Pass event data** box before clicking **OK**. The connection now becomes a solid line.

c. Input property connections:

The following steps create property-to-property connections between the input properties of `Ex02FirstCommand` and the corresponding input property features of the Navigator. These connections are used to propagate the user input values from the property features of the Navigator to `Ex02FirstCommand`.

- Place the cursor on an empty area in the frame window and click the right mouse button. Select **Connect...** and a **Start connection from** [Ex02Navigator] dialog box appears.
- Click **Property**. Select the `FIRST_CMD_IN_LL` property from the scroll-down list and click **OK**.
- Place the cursor on the **Ex02FirstCommand** icon and click the left mouse button. Choose **Connectable Features...** An **End connection to [Ex02FirstCommand]** dialog box appears.
- Click **Property**. Select the `IN_LL` property from the scroll-down list and click **OK**.
- A new property-to-property connection between the `FIRST_CMD_IN_LL` property of the Navigator and the `IN__LL` property of the `Ex02FirstCommand` bean is created.
- Repeat the above steps to associate all of the other features of the `Ex02FirstCommand` with the Navigator:
  - FIRST_CMD_IN_ZZ —> IN__ZZ
  - FIRST_CMD_IN_TRCD —> IN__TRCD
  - FIRST_CMD_IN_CMD —> IN__CMD
  - IN_NAME1 —> IN__NAME1
  - IN_NAME2 —> IN__NAME2
  - IN_EXTN —> IN__EXTN
  - IN_ZIP —> IN__ZIP

d. Result property connections.

   This step connects the output properties of the first command with the Navigator bean so that the results can be accessed and returned to the user. It creates property-to-property connections between the output properties of Ex02FirstCommand and the output property features of the Navigator. These connections are used to propagate the output values from Ex02FirstCommand to the output property features of the Navigator. Because these are result property connections and the result property values are not known until after the command has executed, it is necessary to specify that the trigger event is a change in the value of source property. This is accomplished by selecting the result property-to-property connection after you have created it, right-clicking and selecting **Properties** and then selecting propertyChange as the source event for all result property connections. See the description in Step 5, item 4c, above to create property-to-property connections between the following output properties of the Ex02FirstCommand bean and the corresponding output property features of the Navigator. Remember to start the result property connections by right-clicking on the Ex02FirstCommand icon and end them by clicking in an open area of the Navigator Visual Composition window. Be sure to select propertyChange for the Source event in the **Property-to-property connection - Properties** window.

   - `OUT__LL –> OUT_LL`
   - `OUT__ZZ –> OUT_ZZ`
   - `OUT__MSG –> OUT_MSG`
   - `OUT__CMD –> OUT_CMD`
   - `OUT__NAME1 –> OUT_NAME1`
   - `OUT__NAME2 –> OUT_NAME2`
   - `OUT__EXTN –> OUT_EXTN`
   - `OUT__ZIP –> OUT_ZIP`
   - `OUT__SEGNO –> OUT_SEGNO`

   Save the Ex02FirstCommand bean by clicking on **Bean—>Save Bean**. This will save the Ex02FirstCommand bean and generate the runtime code associated with this bean, some of which will be modified in the next paragraph.

   On your Workbench, expand the Ex02Navigator class and select the connEtoM2 (CommandEvent) method. In the **Source** pane, add the following code in the "user code {1}" block (after the line that reads `// user code begin {1}`):

```
System.out.println( "\nBeginning execution of Ex02FirstCommand ..." + "\n" );
```

   This code will cause the above line to be displayed on the output screen (the Java console) right before Ex02FirstCommand is executed. Save this change by again clicking on the connEtoM2(CommandEvent) method and then click on **Yes**.

Step 6: Create a Command Bean for the Second Command

This step uses the Visual Composition Editor to create a second EAB command and associate to the Navigator. Do the following:

1. Click **Choose Bean...** again. This time the Class Bean Type should be already selected and

com.ibm.connector.ims.sample.phonebook.command.Ex01Command should be highlighted as the Class name. If so, enter Ex02SecondCommand in the Name field and click OK. Otherwise, click **Browse**. In the **Pattern** field enter `Ex01Command`. A list of class names appears. Choose `Ex01Command` (in package `ex01`) for the **Class name** from the list of available classes. Click **OK**. Enter `Ex02SecondCommand` in the **Name** field and click **OK**.

2. To plant the bean, move the cursor to a position in the frame window and click the left mouse button.
3. The following connections are needed for communication between this command, Ex02FirstCommand the Navigator:
   a. UnsuccessfulExecution event connection:

      This step creates an Event-to-method connection such that the returnExecutionUnsuccessful method of the Navigator is invoked when an executionUnsuccessful event of `Ex02SecondCommand` occurs.

      See the description in **Step 5, item 4a** to create an Event-to-method connection between the `executionUnsuccessful(com.ibm.ivj.eab.command.CommandEvent)` event of `Ex02SecondCommand` and the `returnExecutionUnsuccessful(com.ibm.ivj.eab.command.CommandEvent)` method of the Navigator. Be sure to click the **Pass event data** box before clicking **OK** in the **Event-to-method connections - Properties** window.

   b. ReturnSuccessfulExecution event connection:

      This step creates an Event-to-method connection such that the returnExecutionSuccessful method of the Navigator is invoked when an executionSuccessful event of `Ex02SecondCommand` occurs. See the description in **Step 5, item 4a** to create an Event-to-method connection between the `executionSuccessful(com.ibm.ivj.eab.command.CommandEvent)` event of `Ex02SecondCommand` and the `returnExecutionSuccessful(com.ibm.ivj.eab.command.CommandEvent)` method of the Navigator. Be sure to click the **Pass event data** box before clicking **OK**.

   c. Execution event connection:

      This step creates an Event-to-method connection to start the execution of the second command when the first command has executed successfully. The execute method of `Ex02SecondCommand` is invoked when an executionSuccessful event of the `Ex02FirstCommand` occurs.

      - Select the **Ex02FirstCommand** icon and click the right mouse button. Choose **Connect —> Connectable Features...** A **Start connection from [Ex02FirstCommand]** dialog box appears.
      - Click **Event**. Select the `executionSuccessful(com.ibm.ivj.eab.command.CommandEvent)` event from the scroll-down the list and click **OK**.
      - Move the cursor to **Ex02SecondCommand** icon and click the left mouse button. Click **Connectable Features...** An **End connection** dialog box appears.
      - In the **End connection to [Ex02SecondCommand]** dialog box, click **Method**. Choose `execute(com.ibm.ivj.eab.command.CommandEvent)` from the scroll-down list and click **OK**. A new dotted-line connection is created.
      - Select the new connection and click the right mouse button. Select **Properties** from the pop-up menu. An **Event-to-method connection properties** window appears.

- Click the **Pass event data** box and click **OK**. The connection is now a solid line.

d. Properties connections:

   This step sets up the input properties for the `Ex02SecondCommand` bean. In this example, the second command takes input in the form of the `SECOND_CMD_IN_TRCD` and `SECOND_CMD_IN_CMD` which are set in the Java application that runs the Navigator and also from the output of the first command, in this case the `OUT_NAME1` property.

   Note that the properties of Ex02SecondCommand that are not set do not need to be set in this case only because we have set up Ex02SecondCommand to execute an `IVTNO DISPLAY` command. Had we chosen to make Ex02SecondCommand execute another `ADD` command, we would have had to set the other input properties accordingly. Otherwise, those fields in the record added to the database used by `IVTNO` would be blank.

   1) Input Properties from first command:

      This step creates property-to-property connections between the output properties of `Ex02FirstCommand` and the input properties of `Ex02SecondCommand`. These connections are used to propagate the output values from `Ex02FirstCommand` to `Ex02SecondCommand` as its input values.

      - Select the **Ex02FirstCommand** icon and click the right mouse button. Choose **Connect —> Connectable Features...**
      - Click **Property**. Select `OUT_NAME1` property from the scroll-down the list and click **OK**.
      - Place the cursor to **Ex02SecondCommand** icon and click the left mouse button. Click **Connectable Features...** An **End connection to [Ex02SecondCommand]** dialog box appears.
      - In the **End connection to [Ex02SecondCommand]** dialog box, click **Property**. Select `IN_NAME1` property from the scroll-down list and click **OK**. Right-click the connection just created and select Properties. Select propertyChange as the source event in the resulting **Property-to-property connection - Properties** window and click **OK**. A new connection between the `OUT_NAME1` property of the `Ex02FirstCommand` bean and the `IN_NAME1` property of the `Ex02SecondCommand` bean is created.

   2) Input properties from the Navigator bean:

      This step creates property-to-property connections between the input property features of the Navigator and the input properties of `Ex02SecondCommand`. These connections are used to propagate the input values from the input property features of the Navigator to `Ex02SecondCommand` as its input values. See the description above in **Step 5, item c** to create property-to-property connections between the following input property features of the Navigator and the input properties of the `Ex02SecondCommand` bean:

      - `SECOND_CMD_IN_LL –> IN__LL`
      - `SECOND_CMD_IN_ZZ –> IN__ZZ`
      - `SECOND_CMD_IN_TRCD –> IN__TRCD`
      - `SECOND_CMD_IN_CMD –> IN__CMD`

e. Result property connections:

   This step connects the output properties of the second command with the Navigator bean so that the results can be accessed and returned to the user.

It creates property-to-property connections between the output property features of the NavigatorEx02SecondCommand and the properties of Ex02SecondCommandoutput property features of the Navigator and the properties of `Ex02SecondCommand`. These connections are used to propagate the output values from `Ex02SecondCommand` to the output property features of the Navigator. See the description in **Step 5, item c** to create property-to-property connections between the following output properties of the `Ex02SecondCommand` bean and the output property features of the Navigator: Remember to start the result property connections by right-clicking on the Ex02SecondCommand icon and end them by clicking in an open area of the Navigator Visual Composition window. Be sure to select propertyChange for the Source event in the **Property-to-property connection - Properties** window.

- `OUT__LL –> OUT_LL`
- `OUT__ZZ –> OUT_ZZ`
- `OUT__MSG –> OUT_MSG`
- `OUT__CMD –> OUT_CMD`
- `OUT__NAME1 –> OUT_NAME1`
- `OUT__NAME2 –> OUT_NAME2`
- `OUT__EXTN –> OUT_EXTN`
- `OUT__ZIP –> OUT_ZIP`
- `OUT__SEGNO –> OUT_SEGNO`

4. This step adds the `DFS` output message method feature of the second command so that the Navigator bean can display the `DFS` message from the second command.

   Do the following:

   a. Select the **Ex02SecondCommand** icon and click the right mouse button. Choose **Promote Bean Features...** A new window appears.

   b. Ensure that the **Method** radio button is selected and move the GetDFSMessage() method from the left pane to the **Promoted features** section by highlighting and clicking the **>>** button. Click **OK**. A new ex02SecondCommandGetDFSMessage() method is created in the `Ex02Navigator` bean.

Save the Ex02SecondCommand bean by clicking on **Bean—>Save Bean**. This will save the Ex02SecondCommand bean and generate the runtime code associated with this bean, some of which will be modified as described in the next paragraph.

On your Workbench, expand the Ex02Navigator class and select the connEtoM5(CommandEvent) method. In the **Source** pane, add the following code in the "user code {1}" block (after the line that reads `// user code begin {1}`):

```
System.out.println( "The output from Ex02FirstCommand is: " +
                    "\n" +
                    "\nLL: " + getOUT_LL() +
                    "\nZZ: " + getOUT_ZZ() +
                    "\nMSG: " + getOUT_MSG() +
                    "\nCMD: " + getOUT_CMD() +
                    "\nNAME1: " + getOUT_NAME1() +
                    "\nNAME2: " + getOUT_NAME2() +
                    "\nEXTN: " + getOUT_EXTN() +
                    "\nZIP: " + getOUT_ZIP() +
                    "\nSEGNO: " + getOUT_SEGNO() +
```

```
"\n" +
"\n" +
"\nBeginning execution of Ex02SecondCommand..." +
"\n" );
```

This code will cause the above lines to be displayed on the output screen (the Java console) after Ex02FirstCommand has executed successfully but before the start of execution of Ex02SecondCommand. Save this change by again clicking on the connEtoM5(CommandEvent) method and then click on **Yes**.

Step 7: Set Up the Connection Specification for the Navigator (Optional)

This step sets up a separate connection specification for the `Ex02Navigator` bean so that all of the EAB commands in the Navigator can use the same connection specification.

This is an optional step since each EAB command already has its own connection spec. If you choose to create a connection spec in the Navigator, its attributes and methods will override those of all EAB commands that are part of the Navigator. Do the following:

1. In the Visual Composition Editor, click **Choose Bean...** again and click **Browse**. In the **Pattern** field, enter `IMSConnectionSpec`. A list of class names appears. Choose `IMSConnectionSpec` (in package `com.ibm.connector.imstoc`) for the **Class name** from the list of available classes. Click **OK** on this and the previous window.
2. To plant the bean, move the cursor to a position in the frame window and click the left mouse button. By default, it is assigned the name `IMSConnectionSpec1`.
3. To connect the bean to the Navigator, select the **IMSConnectionSpec1** icon and click the right mouse button. Choose **Connect —> this**, then move the cursor to an empty area in the frame window and click the left mouse button. Select `ConnectionSpec` from the **Property** list that appears and click **OK**.
4. Double click the **IMSConnectionSpec1** icon to view the bean properties. Specify the bean properties as follows:

   beanName

   Leave as is.

   Connection timeout

   If you want to provide additional connection properties, see the interface **com.ibm.connector.ConnectionSpecManagementProperties** in package **com.ibm.connector** of the Enterprise Access Builder for a description of this property.

   Host name

   Enter the TCP/IP host name of the machine running IMS Connect that the Navigator (IMS transaction) will be using.

   Maximum connections

   If you want to provide additional connection properties, see the interface **com.ibm.connector.ConnectionSpecManagementProperties** in package **com.ibm.connector** of the Enterprise Access Builder for a description of this property.

   Minimum connections

If you want to provide additional connection properties, see the interface **com.ibm.connector.ConnectionSpecManagementProperties** in package **com.ibm.connector** of the Enterprise Access Builder for a description of this property.

Port

Enter the port for IMS Connect that the Navigator (IMS transaction) will be using.

Reap time

If you want to provide additional connection properties, see the interface **com.ibm.connector.ConnectionSpecManagementProperties** in package **com.ibm.connector** of the Enterprise Access Builder for a description of this property.

Unused timeout

If you want to provide additional connection properties, see the interface **com.ibm.connector.ConnectionSpecManagementProperties** in package **com.ibm.connector** of the Enterprise Access Builder for a description of this property.

Realm

The **Realm** property defines a unique name for a remote target to which the client Java application or servlet is connecting. The **Realm** property is used by the infrastructure LogonInfo of the component server for mapping security information. For example, it maps the client information to the user information. The component server that supports the **Realm** property provides the appropriate value.

5. Optionally, promote the **Host name** and **Port** property features, as follows:
   a. In the Visual Composition Editor, select the **IMSConnectionSpec1** icon and click the right mouse button.
   b. Select **Promote bean feature...** A new window appears.
   c. Ensure that the **Property** radio button is selected and move the **Host name** and **Port** properties from the left pane to the **Promoted features** section by highlighting and clicking the **>>** button. Then click the **OK** button.

Step 8: Promote InteractionSpec Property Features (Optional)

In this step you will promote the dataStoreName and DFSDATA1 bean features of the IMSInteractionSpec beans used by the EAB commands Ex02FirstCommand and Ex02SecondCommand to the interface of the composite Navigator bean.

This is an optional step that you would complete if you want your Java application or servlet to be able to dynamically accept values for and set property features of the IMSInteractionSpec beans used by the EAB commands in your Navigator.

Promote the dataStoreName and DFSDATA1 property features of the IMSInteractionSpec for each of the EAB commands in the Navigator as follows:

1. In the Visual Composition Editor, select the **Ex02FirstCommand** icon and click the right mouse button.
2. Select **Promote bean feature...**. A new window appears.
3. Ensure that the Property radio button is selected and move the dataStoreName and DFSDATA1 property features from the left pane to the **Promoted features** section by highlighting each property feature, and clicking the **>>** button and then clicking the **OK** button.

4. Repeat steps a. b. and c. to promote the dataStoreName and DFSDATA1 properties of the **Ex02SecondCommand** bean.

Step 9: Save the Navigator Bean

Click the **Bean** menu and click **Save Bean** to save your work.

This is the final step in creating the Navigator composite command bean. You can use this bean (Navigator) to run two IMS transactions.

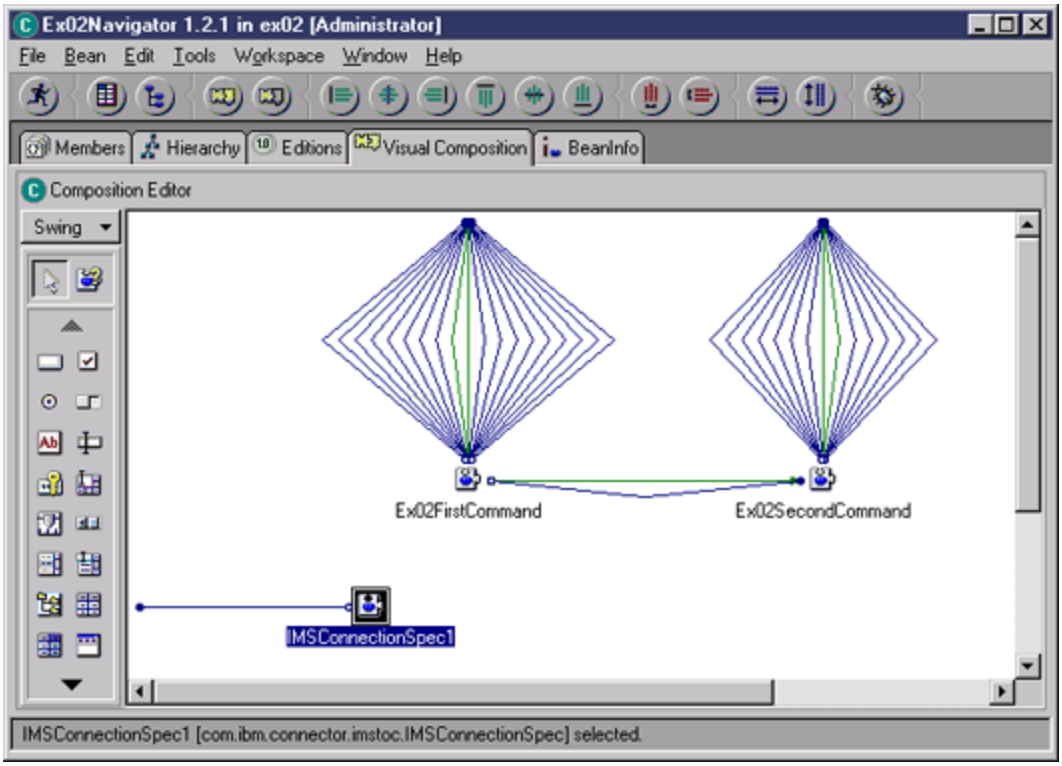


Figure 8. The Completed Navigator Bean as shown in the Visual Composition Editor

Writing a Java Application to Use a Navigator

The following example illustrates how you can use the EAB Navigator created in *Creating a Navigator Using the Command and Visual Composition Editors* to create a Java application. You can use the Java application to run two consecutive IMS transactions.

The steps include the following:

1. Start the Workbench in the VisualAge for Java IDE
2. Create an Executable Class
3. Add Java Code to the Class

Step 1: Start the Workbench in the VisualAge for Java IDE

For a description of how to start the Workbench in the VisualAge for Java IDE, see the documentation referenced in Step 1: Start the Workbench in the VisualAge for Java IDE (page 46) under Writing a Java Application to Use the EAB Command (page 46).

Step 2: Create an Executable Class

For the description of how to create a new executable class, see Step 2: Create an Executable Class (page 46) under Writing a Java Application to Use the EAB Command. (page 46) The sample code for this section uses `Ex02NavigatorExecute` for the name of the executable class.

Step 3: Add Java Code to the Class

Add the following code to the main(String[]) method of the `Ex02NavigatorExecute` class that you created previously. Place the user ID, password, and group name to be sent to IMS in the fields "your UID," your Pwd," and "your Grp," respectively.

```
public static void main(String args[])
{
  //-----------------------------------+
  // Process input arguments.                                          |
  // Each argument will be in the format of: Key=Value                 |
  //                                                                   |
  // The arguments will be processed to get the value for the          |
  // following Keys:                                                   |
  //   HOSTNAME  - the TCP/IP hostname of the machine running IMS      |
  //               TCP/IP OTMA Connection (ITOC) that the command      |
  //               (IMS transaction) will be using.                    |
  //   PORT      - port that IMS Connect          |                    |
  //               (ITOC) will be using on the host machine.           |
  //   DS1       - the name of the target IMS datastore that the      |
  //               first command will be using.                        |
  //   DS2       - the name of the target IMS datastore that the      |
  //               second command will be using.                       |
  //                                                                   |
  // For example, an input argument might be:                          |
  //   HOSTNAME=HOST1 PORT=1234 DS1=MYIMS1 DS2=MYIMS2                  |
  //-----------------------------------+

  int numArgs = 0;
  String hostName   = null; // Use value in IMSConnectionSpec1.
  int    port       = 0;    // Use value in IMSConnectionSpec1.
  String dataStore1 = null; // Use value in IMSInteractionSpec of first command.
  String dataStore2 = null; // Use value in IMSInteractionSpec of second command.

  java.util.Enumeration en = null;

  numArgs = args.length;
  if( numArgs > 0 )
    for( int i = 0; i < numArgs; i++ )
    {
      java.util.StringTokenizer st = new java.util.StringTokenizer( args[i], "=" );
      String key = st.nextToken();
      String val = st.nextToken();
      if( key.equalsIgnoreCase( "HOSTNAME" ) )
      {
        if( val != null )
          hostName = new String( val );
      }
      if( key.equalsIgnoreCase( "PORT" ) )
      {
        if( val != null )
          port = java.lang.Integer.parseInt( val, 10 );
      }
      if( key.equalsIgnoreCase( "DS1" ) )
      {
      if( val != null )
          dataStore1 = new String( val );
      }
```

```
      if( key.equalsIgnoreCase( "DS2" ) )
      {
        if( val != null )
          dataStore2 = new String( val );
      }
    }
  //-------------------------------------+
  // Establish current runtime context.                              |
  //-------------------------------------+
  JavaRuntimeContext runtimeContext = new JavaRuntimeContext();
  JavaRuntimeContext.setCurrent(runtimeContext);

  //-------------------------------------+
  // Set trace level and get a connection manager.                   |
  //-------------------------------------+
  JavaRASService aRASService =
    (JavaRASService)runtimeContext.getRASService();
  aRASService.setTraceLevel(RASService.RAS_TRACE_OFF);
// aRASService.setTraceLevel(RASService.RAS_TRACE_ENTRY_EXIT);
// aRASService.setTraceLevel(RASService.RAS_TRACE_INTERNAL);
  ConnectionManager aConnMgr = new ConnectionManager();
  runtimeContext.setConnectionManager(aConnMgr);

  try
  {
    //-------------------------------+
    // Create an instance of the Navigator bean.     |
    //-------------------------------+
    Ex02Navigator nav = ( Ex02Navigator )java.beans.Beans.instantiate(
                                        null, "com.ibm.connector.ims.sample.phonebook.navigator.Ex02
    if( hostName != null )
    {
      nav.setIMSConnectionSpec1HostName( hostName );
    }
    if( port != 0 )
    {
      nav.setIMSConnectionSpec1PortNumber( port );
    }
    if( dataStore1 != null )
    {
      nav.setEx02FirstCommandDataStoreName( dataStore1 );
    }
    if( dataStore2 != null )
    {
      nav.setEx02SecondCommandDataStoreName( dataStore2 );
    }

    //-------------------------------+
    // Replace values yourUID, yourPwd, and yourGrp in the code    |
    // below with values appropriate to your environment.          |
    //-------------------------------+
    IMSLogonInfoItems logon =
      new IMSLogonInfoItems( runtimeContext.getLogonInfo(),
                             nav.getConnectionSpec() );
    logon.setUser("yourUID");
    logon.setPassword("yourPwd");
    logon.setGroup("yourGrp");
    //-------------------------------+
    // Populate the command bean with input data for the           |
    // IMS transaction.                                            |
    //-------------------------------+
    nav.setFIRST_CMD_IN_LL( (short) ((InMsg)nav.getInput()).getSize() );
    nav.setFIRST_CMD_IN_ZZ( (short)0 );
    nav.setFIRST_CMD_IN_TRCD( "IVTNO" );
    nav.setFIRST_CMD_IN_CMD( "ADD" );
    nav.setIN_NAME1( "JAMES" );
    nav.setIN_NAME2( "BOND" );
```

```
    nav.setIN_EXTN( "007" );
    nav.setIN_ZIP( "12345" );
    nav.setSECOND_CMD_IN_LL( (short) ((InMsg)nav.getInput()).getSize() );
    nav.setSECOND_CMD_IN_ZZ( (short)0 );
    nav.setSECOND_CMD_IN_TRCD( "IVTNO" );
    nav.setSECOND_CMD_IN_CMD( "DISPLAY" );

    //--------------------------------+
    // Invoke the execute method on the command in order to run    |
    // the IMS transaction.                                        |
    //--------------------------------+
    nav.execute();
    //--------------------------------+
    // Get the command's output object (bean) and determine its    |
    // type.                                                      |
    //--------------------------------+
    Object bean = nav.getOutput();
    if( java.beans.Beans.isInstanceOf( bean, ex01.OutMsg.class ) == true )
    {
      //--------------------------------+
      // Retrieve the output of the IMS transaction from the      |
      // command object and print it.                            |
      //--------------------------------+
      System.out.println( "\The output from the Ex02SecondCommand is:" +
                          "\nLL: " + nav.getOUT_LL() +
                          "\nZZ: " + nav.getOUT_ZZ() +
                          "\nMSG: " + nav.getOUT_MSG() +
                          "\nCMD: " + nav.getOUT_CMD() +
                          "\nNAME1: " + nav.getOUT_NAME1() +
                          "\nNAME2: " + nav.getOUT_NAME2() +
                          "\nEXTN: " + nav.getOUT_EXTN() +
                          "\nZIP: " + nav.getOUT_ZIP() +
                          "\nSEGNO: " + nav.getOUT_SEGNO() );
    }
    else
    {
      //-----------------------------+
      // Print message from IMS.                          |
      //-----------------------------+
      System.out.println( "\nMessage from IMS is...\n" +
      nav.ex02SecondCommandGetDFSMessage());
    }
    //--------------------------------+
    // Commit resources.                                          |
    //--------------------------------+
    ((JavaCoordinator)runtimeContext.getCoordinator()).commit();
  }
  catch ( Exception e )
  {
    System.out.println( "\nCaught exception is: " + e );
  }
    //--------------------------------+
    // Close the runtime context.                                 |
    //--------------------------------+
    runtimeContext.close();
}
```

Testing the Java Application

To test the Java application within VisualAge for Java, see Testing Your Application within VisualAge for Java (page 50).

Running the Java Application

To run the Java application with the Navigator outside of VisualAge for Java, see Running Your Application Outside of VisualAge for Java. (page 50)

Chapter 6. Building a Java Application for an IMS Transaction with Multi-Segment Output Messages

This section describes how to build a Java application that processes an IMS transaction that has a single-segment input message and an output message with multiple segments.

This example corresponds to one of the samples included in the IMS Connector Samples feature. The Java source relating to this example is contained in the com.ibm.connector.ims.sample.multisegout package of the Connector IMS Samples project which is added to your workspace when the IMS Connector Samples feature is added. See Preparing your VisualAge for Java Environment for instructions on installing the IMS Connector Samples feature if you have not already done so. By following the instructions in this section, you will create an application which is the same as the corresponding sample except that it uses different project and package names than are used in the sample since you can't have duplicate package or project names within your VisualAge environment. As a result, you will be able to check your work against the sample, helping you to resolve any difficulties you encounter as you progress through the steps that follow.

Attention: The Java application might not be executable, because the corresponding IMS application program might not be available in all environments.

The IMS application program used with the sample code in this section is DFSDDLT0. The DFSDDLT0 script for this sample, MSOut.scr is provided in the <*IBM_Connectors_install_dir*> \imsconn\samples\misc.

The following stage 1 source was used in the IMS environment of this example:

```
APPLCTN   PSB=STLDDLT2, PGMTYPE=(,,2)
TRANSACT  CODE=SKS2, PRTY=(8,8)
```

The PSB and CODE values can be changed to match your IMS system definition requirements. The PSB requires only an I/O PCB. It does not require alternate PCBs or database PCBs.

Related Reading: For information on using DFSDDLT0, see the *IMS Application Programming Guide*.

The processing of a multiple segment output message differs from that of a single segment output message in that the transaction output is represented by a class that implements the IByteBuffer interface of the IBM Java Record Library, rather than a transaction output message bean created by the VisualAge for Java Record Generator. The IMS Connector for Java populates the byte array of the output class with the transaction output message. The Java application extracts the individual segments from the byte array and uses them to set the values of the output segment objects. The output segment objects are created using the VisualAge for Java Record Generator from COBOL 01 descriptors of the individual message segments.

Related Reading: For information on building a Java application that processes an IMS transaction that has an input message with multiple segments and a single-segment output message, see Building a Java Application for an IMS Transaction with Multi-Segment Input Messages.

© Copyright IBM Corp. 1998, 2000

A method for building a Java application for an IMS transaction with multi-segment output messages is covered by the following sections:

1. Creating an EAB Command Using the Command Editor
2. Writing a Java Application to Use the EAB Command
3. Testing Your Application within VisualAge for Java
4. Running Your Application outside of VisualAge for Java

Some of the steps in the following sections are not described in detail, since they are covered by earlier sections.

Creating an EAB Command Using the Command Editor

This section describes how you create an EAB command for an IMS transaction with a multi-segment output message using the Command Editor.

The steps include the following:

1. Start the Workbench in the VisualAge for Java IDE
2. Create a Project and Package for the Application
3. Create a COBOL RecordType Class for the IMS Transaction Input Message
4. Create a Transaction Input Message Record Bean
5. Create a Transaction Output Message Class that Implements IByteBuffer
6. Create COBOL RecordType Classes for the Output Message Segments
7. Create Record Beans for the Output Message Segments
8. Create a Command Class for the IMS Transaction
9. Set Up the Interaction Specification for the Command
10. Set Up the Connection Specification for the Command
11. Set Up the Command Input
12. Set Up the Command Output
13. Create the Command Bean

Step 1: Start the Workbench in the VisualAge for Java IDE

For a description of how to start the Workbench in the VisualAge for Java IDE, see Step 1: Start the Workbench in the VisualAge for Java IDE (page 36) under Creating IMS Transaction Input and Output Beans (page 36).

Step 2: Create a Project and Package for the Application

For a description of this step, see Step 2: Create a Project and Package for the Application (page 36) under Creating IMS Transaction Input and Output Beans (page 36). The package used by the sample code for this section is **multisegout**.

Step 3: Create a COBOL RecordType Class for the IMS Transaction Input Message

For a description of how to create a COBOL RecordType class from the COBOL source for an IMS input message, see Step 3: Create a COBOL RecordType Class for the IMS Transaction Input Message (page 37) under Creating IMS Transaction Input and Output Beans (page 36). The COBOL source for the input message can be found in file MSOut.ccp in \imsconn\samples\misc, where the IBM Connectors installation directory is located.

Step 4: Create a Transaction Input Message Record Bean

For a description of how to create a transaction input message record bean from a COBOL RecordType class, see Step 4: Create a Transaction Input Record Bean (page 38) under Creating IMS Transaction Input and Output Beans (page 36). The sample code for this section uses `InMsg` for the name of the input message record bean.

Step 5: Create a Transaction Output Message Class that Implements IByteBuffer

This step creates a class that implements the **IByteBuffer** interface of the package com.ibm.record of the IBM Java Record Library.

1. With the package **multisegout** selected, click the **Selected** menu, then select **Add—>Class...** The **Create Class** wizard appears.
2. Ensure that the correct names appear in the **Project** and **Package** entry fields.
3. Select the **Create a new class** radio button. Enter a name for the new class (for example, `AllSegs`), and ensure that the superclass is **java.lang.Object**.
4. Ensure that the **Browse the class when finished** and the **Compose the class visually** check boxes are *not* checked.
5. Select the **Next** button. The **Attributes** wizard appears.
6. Enter `com.ibm.record.IByteBuffer` in the **Which interfaces should this class implement?** list.
7. Check the **public** check box and the recommended check boxes under **Which method stubs would you like to create?**.
8. Select the **Finish** button to create the class.
9. Modify the code generated for the class by VisualAge for Java as follows:
   a. Add the following private field to the `AllSegs` class:
      ```
      private byte[] bytes;
      ```
   b. Modify the checkBytes method so that it returns true instead of false.
      ```
      return true;
      ```
   c. Modify the getBytes method so that it returns the private field bytes.
      ```
       return bytes;
      ```
   d. Modify the setBytes method so that it sets the private field bytes.
      ```
      bytes = arg1;
      ```

Step 6: Create COBOL RecordType Classes for the Output Message Segments

For a description of how to create a COBOL RecordType class from COBOL source, see Step 5: Create a COBOL RecordType Class for the IMS Transaction Output Message (page 39) under Creating IMS Transaction Input and Output Beans

(page 36). Create a COBOL RecordType for each segment of the output message. The COBOL source for the output message segments can be found in file **MSOut.ccp** in <*IBM_Connectors_install_dir*>\imsconn\samples\misc, where <IBM_Connectors_install_dir> is the IBM Connectors installation directory.

Step 7: Create Record Beans for the Output Message Segments

For a description of how to create a record bean from a COBOL RecordType class, see Step 4: Create a Transaction Input Record Bean (page 38) under Creating IMS Transaction Input and Output Beans (page 36). Create three record beans, `OutSeg1`, `OutSeg2`, and `OutSeg3`.

Step 8: Create a Command Class for the IMS Transaction

For a description of how to create a command class, see Step 1: Create a Command Class for the IMS Transaction (page 40) under Creating an EAB Command Using the Command Editor (page 39). The command class used by the sample code for this section is MSOutCommand. After completing this step, you will be working in the Visual Composition Editor.

Step 9: Set Up the Interaction Specification for the Command

For a description of how to set up the Interaction Specification for the command, see Step 5: Promote the Interaction Specification Properties (page 44) under Creating an EAB Command Using the Command Editor (page 39). This includes adding the IMSInteractionSpec bean, connecting it to the EAB command, and promoting the Datastore Name property.

Step 10: Set Up the Connection Specification for the Command

For a description of how to set up the Connection Specification for the command, see Step 6: Promote the Connection Specification Properties (page 45) under Creating an EAB Command Using the Command Editor (page 39). This includes adding the IMSConnectionSpec bean, connecting it to the EAB command, and promoting the **Host name** and **Port** properties.

Step 11: Set Up the Command Input

For a description of how to set up the command input, see *Writing a Java Application to Use the EAB Command*, detailed below. This includes adding the InMsg bean, connecting it to the EAB command, and promoting the **LL**, **ZZ**, **TRCD**, **DATA1**, and **DATA2** properties.

Step 12: Set Up the Command Output

For a description of how to set up the IMS transaction output, see Testing Your Application within VisualAge for Java (page 50). In this case, the class that implements IByteBuffer (AllSegs) is associated with the command output. After adding the AllSegs bean and connecting it to the command, promote the bytes property of the bean.

Step 13: Create the Command Bean

For a description of how to create the composite EAB command bean, see Step 7: Create the Command Bean (page 45) under Creating an EAB Command Using the Command Editor (page 39).

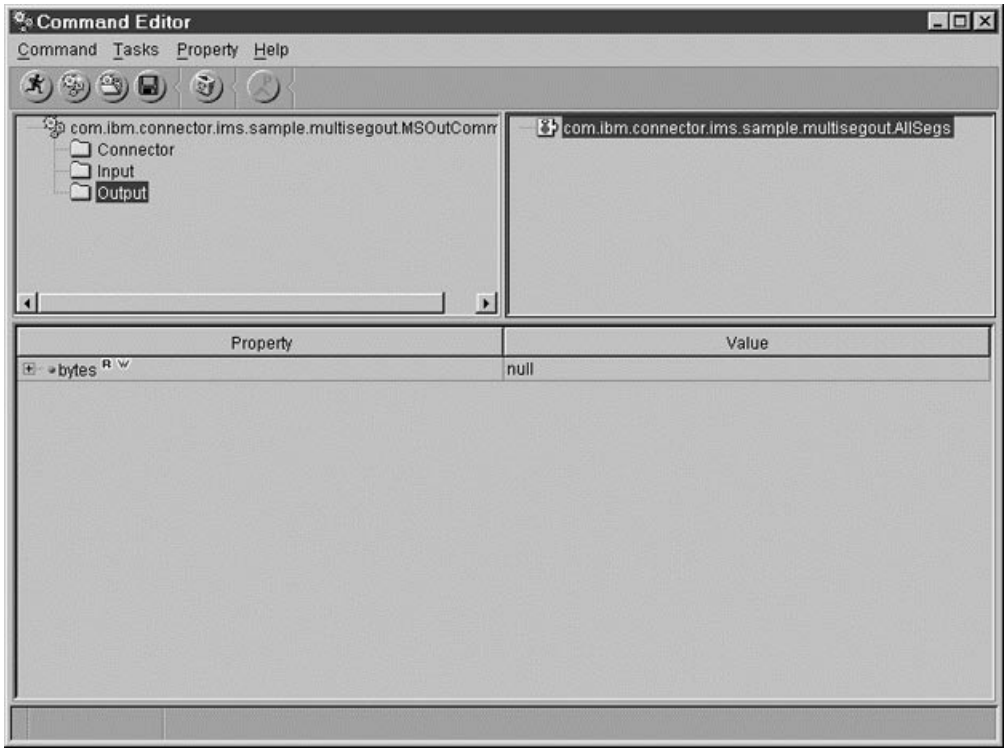


Figure 9. The Completed EAB Command as shown in the Visual Composition Editor

Chapter 7. Building a Java Application for an IMS Transaction with Multi-Segment Input Messages

This section describes how to build a Java application that processes an IMS transaction that has a multiple-segment input message and a single-segment output message.

This example corresponds to one of the samples included in the IMS Connector Samples feature. The Java source relating to this example is contained in the com.ibm.connector.ims.sample.multisegin package of the Connector IMS Samples project which is added to your workspace when the IMS Connector Samples feature is added. See Preparing your VisualAge for Java Environment for instructions on installing the IMS Connector Samples feature if you have not already done so. By following the instructions in this section, you will create an application which is the same as the corresponding sample except that it uses different project and package names than are used in the sample since you can't have duplicate package or project names within your VisualAge environment. As a result, you will be able to check your work against the sample, helping you to resolve any difficulties you encounter as you progress through the steps that follow.

Attention: The Java application might not be executable, because the corresponding IMS application program might not be available in all environments.

The IMS application program used with the sample code in this section is DFSDDLT0. The DFSDDLT0 script for this sample, **MSIn.scr**, can be found in <*IBM_Connectors_install_dir*>\imsconn\ samples\misc

The following stage 1 source was used in the IMS environment of this example:

```
APPLCTN   PSB=STLDDLT2, PGMTYPE=(,,2)
TRANSACT  CODE=SKS2, PRTY=(8,8)
```

The PSB and CODE values can be changed to match your IMS system definition requirements. The PSB requires only an I/O PCB. It does not require alternate PCBs or database PCBs.

Related Reading: For information on using DFSDDLT0, see the *IMS Application Programming Guide*.

The process for processing a multiple-segment input message differs from that of a single-segment input message, in that the transaction input is represented by a class that implements the **IByteBuffer** interface of the IBM Java Record Library, rather than a transaction input message bean created by the VisualAge for Java Record Generator. The Java application populates individual input segment objects, then builds the transaction input buffer. The input segment objects are created using the VisualAge for Java Record Generator from COBOL 01 descriptors of the individual message segments. A method for building a Java application for an IMS transaction with multiple-segment input messages is covered in the following sections:

1. Creating an EAB Command Using the Command Editor
2. Writing a Java Application to Use the EAB Command
3. Testing Your Application within VisualAge for Java
4. Running Your Application outside of VisualAge for Java

© Copyright IBM Corp. 1998, 2000

Several of the steps in the sections that follow are not described in detail, because they are covered in earlier sections.

Creating an EAB Command Using the Command Editor

This section describes how you create an EAB command for an IMS transaction with a multi-segment input message using the Command Editor. The steps include the following:

1. Start the Workbench in the VisualAge for Java IDE
2. Create a Project and Package for the Application
3. Create COBOL RecordType Classes for the Input Message Segments
4. Create Record Beans for the Input Message Segments
5. Create a Transaction Input Message Class that Implements IByteBuffer
6. Create a COBOL RecordType Class for the IMS Transaction Output Message
7. Create a Transaction Output Message Record Bean
8. Create a Command Class for the IMS Transaction
9. Promote the IMS Transaction Input Properties
10. Promote the IMS Transaction Output Properties
11. Promote the DFS Message Properties
12. Promote the Interaction Specification Properties
13. Promote the Connection Specification Properties
14. Create the Command Bean

Step 1: Start the Workbench in the VisualAge for Java IDE

For a description of how to start the Workbench in the VisualAge for Java IDE, see Step 1: Start the Workbench in the VisualAge for Java IDE (page 36) under Creating IMS Transaction Input and Output Beans (page 36).

Step 2: Create a Project and Package for the Application

For a description of this step, see Step 2: Create a Project and Package for the Application (page 36) under Creating IMS Transaction Input and Output Beans (page 36). The package used by the sample code for this section is **multisegin**.

Step 3: Create COBOL RecordType Classes for the Input Message Segments

For a description of how to create a COBOL RecordType class from the COBOL source for an IMS input message, see Step 3: Create a COBOL RecordType Class for the IMS Transaction Input Message (page 37) under Creating IMS Transaction Input and Output Beans (page 36). Create a COBOL RecordType for each segment of the input message. The COBOL source for the input message segments can be found in file **MsIn.cpp**, located in `<IBM_Connectors_install_dir>\imsconn\samples\misc`, where *<IBM_Connectors_install_dir>* is the IBM Connectors installation directory. For example, `d:\IBM Connectors`. Initially, <IBM_Connectors_install_dir> is **<vaj_install_drive>\IBM Connectors**, where `<vaj_install_drive>` is the drive on which VisualAge for Java is installed.

Step 4: Create Record Beans for the Input Message Segments

For a description of how to create a record bean from a COBOL RecordType class, see Step 4: Create a Transaction Input Record Bean (page 38) under Creating IMS Transaction Input and Output Beans (page 36). Create four record beans: `InSeg1`, `InSeg2`, `InSeg3`, and `InSeg4`.

Step 5: Create a Transaction Input Message Class that Implements IByteBuffer

This step creates a class that implements the IByteBuffer interface of the package com.ibm.record of the IBM Java Record Library.

1. With the package **multisegin** selected, click the **Selected** menu, then select **Add—>Class...** The **Create Class** wizard appears.
2. Ensure that the correct names appear in the **Project** and **Package** entry fields.
3. Select the **Create a new class** radio button. Enter a name for the new class (for example, `AllInSegs`), and ensure that the superclass is **java.lang.Object**.
4. Ensure that the **Browse the class when finished** and the **Compose the class visually** check boxes are *not* checked.
5. Select the **Next** button. The **Attributes** wizard appears.
6. Enter `com.ibm.record.IByteBuffer` in the **Which interfaces should this class implement?** list.
7. Check the **public** check box and the recommended check boxes under **Which method stubs would you like to create?**.
8. Select the **Finish** button to create the class.
9. Modify the code generated for the class by VisualAge for Java as follows:
   a. Add the following private field to the `AllInSegs` class:
      ```
      private byte[] bytes;
      ```
   b. Modify the checkBytes method so that it returns true instead of false.
      ```
      return true;
      ```
   c. Modify the getBytes method so that it returns the private field bytes.
      ```
       return bytes;
      ```
   d. Modify the setBytes method so that it sets the private field bytes.
      ```
      bytes = arg1;
      ```

Step 6: Create a COBOL RecordType Class for the IMS Transaction Output Message

For a description of how to create a COBOL RecordType class from the COBOL source for an IMS output message, see Step 5: Create a COBOL RecordType Class for the IMS Transaction Output Message (page 39) under Creating IMS Transaction Input and Output Beans (page 36).

The COBOL source for the output message can be found in file **MsIn.cpp**, located in `<IBM_Connectors_install_dir>\imsconn\samples\misc`, where *<IBM_Connectors_install_dir>* is the IBM Connectors installation directory.

Step 7: Create a Transaction Output Message Record Bean

For a description of how to create a transaction output message record bean from a COBOL RecordType class, see Step 6: Create a Transaction Output Record Bean (page 39) under Creating IMS Transaction Input and Output Beans (page 36). The sample code for this section uses `OutMsg` for the name of the output message record bean.

Step 8: Create a Command Class for the IMS Transaction

For a description of how to create a command class, see Step 1: Create a Command Class for the IMS Transaction (page 40) under Creating an EAB Command Using the Command Editor (page 39). Briefly, use the Command Editor to create an EAB command with name MSInCommand that includes an interaction specification of IMSInteractionSpec, a connection specification of IMSConnectionSpec, the AllInSegs input record bean, and output record beans of OutMsg and DFSMsg.

Step 9: Promote the IMS Transaction Input Properties

For a description of how to promote properties of an input record bean, see Step 2: Promote the IMS Transaction Input Properties (page 42) under Creating an EAB Command Using the Command Editor (page 39). In this case, promote the bytes property of the AllInSegs Input record bean.

Step 10: Promote the IMS Transaction Output Properties

For a description of how to promote properties of an output record bean, see Step 3: Promote the IMS Transaction Output Properties (page 43) under Creating an EAB Command Using the Command Editor (page 39). In this case, promote the LL, ZZ, and DATA1, properties of the OutMsg Output record bean.

Step 11: Promote the DFS Message Properties

For a description of how to promote properties of the DFSMsg output record bean, see Step 4: Promote the DFS Message Properties (page 43) under Creating an EAB Command Using the Command Editor (page 39). In this case, promote the methods getDFSMessage and getDFSMessageSegments, as well as the DFSDATA1 property of the DFSMsg output record bean.

Step 12: Promote the Interaction Specification Properties

For a description of how to promote properties of an interaction specification, see Step 5: Promote the Interaction Specification Properties (page 44) under Creating an EAB Command Using the Command Editor (page 39). In this case, promote the Datastore Name property of the IMSInteractionSpec record bean.

Step 13: Promote the Connection Specification Properties

For a description of how to promote properties of an connection specification, see Step 6: Promote the Connection Specification Properties (page 45) under Creating an EAB Command Using the Command Editor (page 39). In this case, promote the Host name and Port properties of the IMSConnectionSpec record bean.

Step 14: Create the Command Bean

For a description of how to save the command bean, see Step 7: Create the Command Bean (page 45) under Creating an EAB Command Using the Command Editor (page 39).

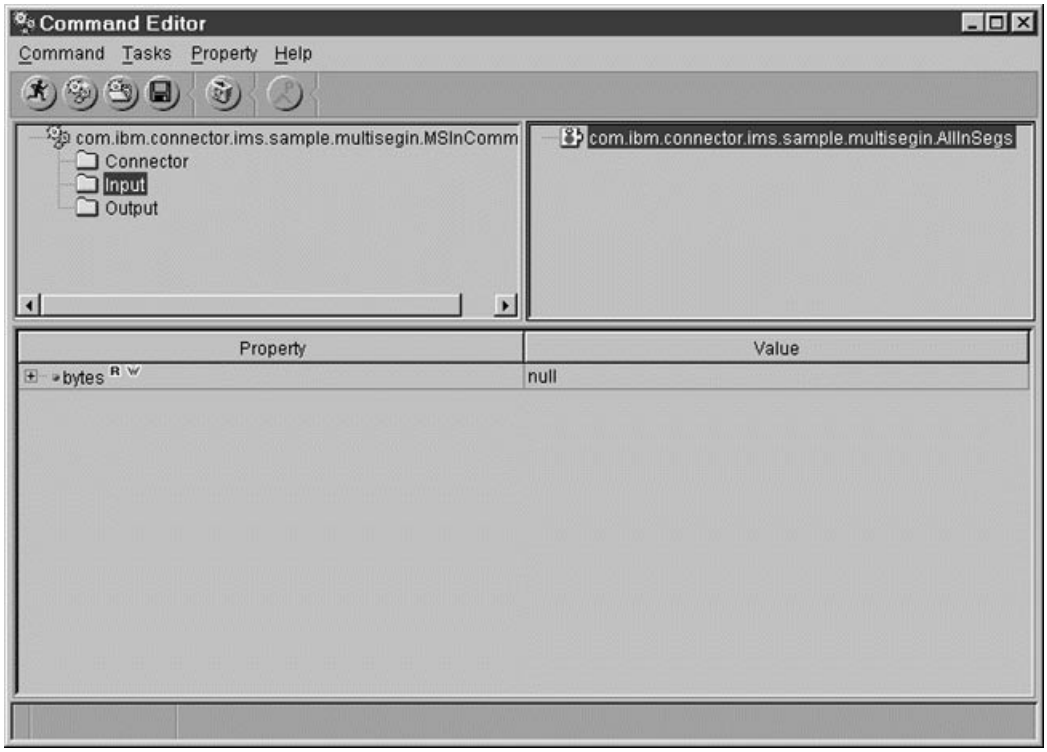


Figure 10. The Completed EAB Command as shown in the Command

Chapter 8. Building an Application to Run an IMS Transaction with Synchronization Level Confirm

This section illustrates how you can build a Java application that runs an IMS transaction with a synchronization level of **Confirm**.

The instructions in this section demonstrate on how to create EAB commands with MODE_ACK and MODE_NACK that send positive and negative acknowledgements to IMS when the coordinator used by the runtime context is not coordinating (e.g., NullCoordinator).

When a coordinator being used by the runtime context is coordinating (e.g., JavaCoordinator), it is recommended to use the commit() and rollback() methods of the coordinator to send positive and negative acknowledgements to IMS. The source code for such example is shipped as the class Ex04ExecuteWithCoordinator included in the com.ibm.connector.ims.sample.phonebook.confirm package of the Connector IMS Sample project.

For information on the synchronization level of an interaction of a command and the different scenarios to build a Java application that runs an IMS transaction with synchronization level Confirm, see Synchronization Level (page 11) section in IMS Connector for Java concepts and terms.

This example uses the IMS transaction that is described in the IMS INSTALL/IVP Sample Application, and corresponds to one of the samples included in the IMS Connector Samples feature. The Java source relating to this example is contained in the com.ibm.connector.ims.sample.phonebook.confirm package of the Connector IMS Samples project which is added to your workspace when the IMS Connector Samples feature is added. See Preparing your VisualAge for Java Environment for instructions on installing the IMS Connector Samples feature if you have not already done so. By following the instructions in this section, you will create an application which is the same as the corresponding sample, with the exception that your sample uses different project and package names than are used in the sample since you can't have duplicate package or project names within your VisualAge environment. As a result, you will be able to check your work against the sample, helping you to resolve any difficulties you encounter as you progress through the steps that follow.

Creating the MODE_ACK and MODE_NACK EAB commands

The following steps create commands to run an IMS transaction with Synchronization level Confirm when using
MODE_ACK and MODE_NACK commands to send positive and negative acknowledgements to IMS:

1. Start the Workbench in the VisualAge for Java IDE
2. Create a Project and Package for the Application
3. Create a Command to Execute the IMS Transaction
4. Create a Command for the MODE_ACK Interaction
5. Create a Command for the MODE_NACK Interaction

Step 1: Start the Workbench in the VisualAge for Java IDE

© Copyright IBM Corp. 1998, 2000

For a description of how to start the Workbench in the VisualAge for Java IDE, see Step 1: Start the Workbench in the VisualAge for Java IDE (page 36) under Creating IMS Transaction Input and Output Beans (page 36) in Building a Java Application to Run an IMS Transaction.

Step 2: Create a Project and Package for the Application

For a description of this step, see Step 2: Create a Project and Package for the Application (page 36) under Creating IMS Transaction Input and Output Beans (page 36). The package used by the sample code for this section is `ex04`.

Step 3: Create a Command to Execute the IMS Transaction

This step creates a command for an IMS transaction with a synchronization level of **Confirm**.

1. Create an EAB command with the name, for example, Ex04Command.
2. Edit the interaction specification properties of the command.
3. Select SYNC_LEVEL_CONFIRM for the property Synchronization Level.

Related Reading: For information on how to create and set up a command, see Step 1: Creating a Command Class for the IMS Transaction (page 40) under Creating an EAB Command Using the Command Editor (page 39).

Step 4: Create a Command for the MODE_ACK Interaction

This step creates a command for the MODE_ACK interaction with IMS. The MODE_ACK command is used to send a positive acknowledgment to IMS. It informs IMS that the output data has been received and tells IMS to commit the transaction.

1. Create an EAB command with the name, for example, Ex04AckCommand.

   Since the MODE_ACK command is not a transaction request, it does not contain any user input or output data. As a result, the MODE_ACK command consists only of the IMSInteractionSpec and IMSConnectionSpec objects. The MODE_ACK command does not contain any input or output objects.
2. Edit the interaction specification properties of the command.

   Select MODE_ACK for the Mode property.

Related Reading: For information on how to create a Command class, see Step 1: Creating a Command Class for the IMS Transaction (page 40) under Creating an EAB Command Using the Command Editor (page 39).

Step 5: Create a Command for the MODE_NACK Interaction

This step creates a command for the MODE_NACK interaction with IMS. The MODE_NACK command is used to send a negative acknowledgment to IMS. It notifies IMS that the user is not satisfied with the output data, and requests IMS to roll back the transaction.

1. Create an EAB command with the name, for example, Ex04NackCommand.

   Since the MODE_NACK command is not a transaction request, it does not contain any user input or output data. As a result, the MODE_NACK command consists only of the IMSInteractionSpec and IMSConnectionSpec objects. The MODE_NACK command does not contain any input or output objects.

2. Edit the interaction specification properties of the command.

   Select MODE_NACK for the Mode property.

Writing a Java Application to Run the Transaction

Step 1: Start the Workbench in the VisualAge for Java IDE

For a description of how to start the Workbench in the VisualAge for Java IDE, see Step 1: Start the Workbench in the VisualAge for Java IDE (page 46) under Writing a Java Application to Use the EAB Command (page 46) in Building a Java Application to Run an IMS Transaction.

Step 2: Create an Executable Class

This step creates an executable class for this example.

For a description on how to create an Executable Class using the package created in *Step 2: Create a Project and Package for the Application* above, see Step 2: Create an Executable Class (page 46). Specify `Ex04Execute` as the name of the class, and a new class named Ex04Execute will be created.

Step 3: Add Java Code to the Class

This step adds the code below to the main(String[]) method of the Ex04Execute class created previously. Replace the values for user ID, password, and group name, according to your IMS environment.

```
//-------------------------------------+
// Run IMS Installation Verification Test application with a          |
// synchronization level of Confirm.                                  |
//-------------------------------------+
public static void main(java.lang.String[] args)
{
  int     numArgs         = 0;
  String  hostName        = null; // Use value in IMSConnectionSpec.
  int     port            = 0;    // Use value in IMSConnectionSpec.
  String  dataStore       = null; // Use value in IMSInteractionSpec.
  //-------------------------------------+
  // Process input arguments.                                         |
  // Each argument will be in the format of:                          |
  //   Key=Value                                                      |
  //                                                                  |
  // The arguments will be processed to get the value for the         |
  // following Keys:                                                  |
  //   HOST NAME  - the TCP/IP hostname of the machine running IMS    |
  //                Connect that the command                          |
  //                (IMS transaction) will be using.                  |
  //   PORT      - port of the IMS Connect                            |
  //                 will be using on the host machine.               |
  //   DATASTORE - the name of the target IMS datastore               |
  //                                                                  |
  // For example, to supply 'host1' as the Host name the       |
  // argument should be in the format:                                |
  //   HOSTNAME=host1                                                 |
  //-------------------------------------+
  numArgs = args.length;
  if( numArgs > 0 )
    for( int i = 0; i < numArgs; i++ )
    {
      java.util.StringTokenizer st = new java.util.StringTokenizer( args[i], "=" );
      String key = st.nextToken();
```

```
      String val = st.nextToken();
      if( key.equalsIgnoreCase( "HOSTNAME" ) )
      {
        if( val != null )
        {
          hostName = new String( val );
        }
      }
      if( key.equalsIgnoreCase( "PORT" ) )
      {
        if( val != null )
        {
          port = java.lang.Integer.parseInt( val, 10 );
        }
      }
      if( key.equalsIgnoreCase( "DATASTORE" ) )
      {
        if( val != null )
        {
          dataStore = new String( val );
        }
      }
    }
//-----------------------------------+
// Use the default Runtime Context that is not coordinating          |
// (i.e. using NullCoordinator).                                     |
//-----------------------------------+
RuntimeContext runtimeContext = RuntimeContext.getCurrent();
ConnectionManager aConnMgr = new ConnectionManager();
runtimeContext.setConnectionManager(aConnMgr);
 try
 {
   //---------------------------------+
   // Create an instance of the command bean.                    |
   //---------------------------------+
   ex04.Ex04Command cmd =
      (ex04.Ex04Command)java.beans.Beans.instantiate(null, ex04.Ex04Command");
   if(hostName != null)
   {
     cmd.setHostName(hostName);
   }
   if( port != 0 )
   {
     cmd.setPortNumber(port);
   }
   if(dataStore != null)
   {
     cmd.setDataStoreName(dataStore);
   }
   //-----------------------------+
   // Provide your environment's values for user ID,       |
   // password and group name for the runtime context.     |
   //-----------------------------+
   IMSLogonInfoItems logon =
     new IMSLogonInfoItems( runtimeContext.getLogonInfo(),
                            cmd.getConnectionSpec() );
   logon.setUser("yourUID");
   logon.setPassword("yourPwd");
   logon.setGroup("yourGrp");
   //---------------------------------+
   // Populate the command bean with input data for the          |
   // IMS transaction.                                           |
   //---------------------------------+
   cmd.setIN__LL( (short)59 );
   cmd.setIN__ZZ( (short)0 );
   cmd.setIN__TRCD(  "IVTNO" );
   cmd.setIN__CMD(   "DISPLAY" );
```

```
    cmd.setIN__NAME1( "Bond" );
    cmd.setIN__NAME2( "" );
    cmd.setIN__EXTN(  "" );
    cmd.setIN__ZIP(   "" );
    //---------------------------------+
    // Invoke the execute method on the command in order to run    |
    // the IMS transaction.                                        |
    //---------------------------------+
    cmd.execute();
    //---------------------------------+
    // Get the command's output object (bean) and determine its    |
    // type.                                                       |
    //---------------------------------+
    Object bean = cmd.getOutput();
    if( java.beans.Beans.isInstanceOf( bean, ex04.OutMsg.class ) == true )
    {
      //-------------------------------+
      // Retrieve the output of the IMS transaction from the      |
      // command object and print it.                            |
      //-------------------------------+
      System.out.println( "\nOuput from IVTNO is..." +
                          "\nLL:    " + cmd.getOUT__LL()    +
                          "\nZZ:    " + cmd.getOUT__ZZ()    +
                          "\nMSG:   " + cmd.getOUT__MSG()   +
                          "\nCMD:   " + cmd.getOUT__CMD()   +
                          "\nNAME1: " + cmd.getOUT__NAME1() +
                          "\nNAME2: " + cmd.getOUT__NAME2() +
                          "\nEXTN:  " + cmd.getOUT__EXTN()  +
                          "\nZIP:   " + cmd.getOUT__ZIP()   +
                          "\nSEGNO: " + cmd.getOUT__SEGNO() );
    }
    else if( java.beans.Beans.isInstanceOf( bean, DFSMsg.class ) == true )
    {
      //-----------------------------+
      // Print message from IMS.                                |
      //-----------------------------+
      System.out.println( "\nMessage from IMS is...\n" +
                           cmd.getDFSDATA1() );
    }
    //---------------------------------+
    // Commit or Rollback the transaction                          |
    // Provide the user logic to decide to commit the transaction and |
    // uncomment out the following block of code.                  |
    //---------------------------------+
/*  if (SOME USER LOGIC THAT DECIDES TO COMMIT THE TRANSACTION)
    {
      //---------------------------------+
      // Create an instance of the command bean.                  |
      //---------------------------------+
      Ex04AckCommand ackCmd =
        (Ex04AckCommand)java.beans.Beans.instantiate(null, "ex04.Ex04AckCommand");
      if(hostName != null)
      {
        ackCmd.setHostName(hostName);
      }
      if( port != 0 )
      {
        ackCmd.setPortNumber(port);
      }
      if(dataStore != null)
      {
        ackCmd.setDataStoreName(dataStore);
      }
      ackCmd.execute();
    }
    else // Rollback the transaction
    {
```

```
      //-------------------------------+
      // Create an instance of the NACK command bean.               |
      //-------------------------------+
      Ex04NackCommand nackCmd =
        (Ex04NackCommand)java.beans.Beans.instantiate(null, "ex04.Ex04NackCommand");
      if(hostName != null)
      {
        nackCmd.setHostName(hostName);
      }
      if( port != 0 )
      {
        nackCmd.setPortNumber(port);
      }
      if(dataStore != null)
      {
        nackCmd.setDataStoreName(dataStore);
      }
      nackCmd.execute();
    }
*/
    }
  catch ( Exception e )
  {
    System.out.println( "\nCaught exception is: " + e );
  }
    //-------------------------------+
    // Close the run-time context.                                  |
    //-------------------------------+
    runtimeContext.close();
    runtimeContext.removeCurrent();
  }
```

Testing the Java Application

To test the Java application within VisualAge for Java, see Testing Your Application within VisualAge for Java (page 50)..

Running the Java Application

To run the Java application with the Navigator outside of VisualAge for Java, see Running Your Application Outside of VisualAge for Java. (page 50)

Chapter 9. Building the Graphical User Interface

This example illustrates how you can use the EAB command that you created in Creating an EAB Command Using the Command Editor (page 39) to create a GUI Java application. You can use the Java application to run an IMS transaction. This example illustrates only basic concepts, and does not include sophisticated GUI input and output processing.

This example corresponds to one of the samples included in the IMS Connector Samples feature. The Java source relating to this example is contained in the com.ibm.connector.ims.sample.phonebook.gui package of the Connector IMS Samples project which is added to your workspace when the IMS Connector Samples feature is added. See Preparing your VisualAge for Java Environment for instructions on installing the IMS Connector Samples feature if you have not already done so. By following the instructions in this section, you will create an application which is the same as the corresponding sample except for the fact that your application will use different project and package names than those that are used in the sample, since package or project names cannot be duplicated within your VisualAge environment. As a result, you will be able to check your work against the sample, helping you to resolve any difficulties you encounter as you progress through the steps that follow.

The steps include the following:

1. Start the Workbench in the VisualAge for Java IDE
2. Create the Main GUI Dialog Class
3. Add the Import Statements
4. Add Connection Manager and Run-Time Context
5. Create the Main GUI Dialog Interface
6. Create a Command to Execute the IMS Transaction
7. Set Up the IMS Command
8. Finish

Step 1: Start the Workbench in the VisualAge for Java IDE

For a description of how to start the Workbench in the VisualAge for Java IDE, see Step 1: Start the Workbench in the VisualAge for Java IDE (page 36) under Creating IMS Transaction Input and Output Beans (page 36) in Building a Java Application to Run an IMS Transaction. The package used by the sample code for this section is `ex03`.

Step 2: Create the Main GUI Dialog Class

From the **Selected** menu, click **Add —> Class**. A wizard appears to request all of the necessary information to create a class:

1. Ensure that the **Project** field contains your project name, `Examples`.
2. Ensure that the **Package** field contains your package name, `ex03`.
3. Click **Create a new class**.
4. Enter `Ex03MainDialog` in the **Class name** field.
5. To choose the superclass for class `Ex03MainDialog`, click **Browse**, then select **JDialog** from the **TypeNames** list.
6. Ensure that **javax.swing** is selected as the package name and click **OK**.

© Copyright IBM Corp. 1998, 2000

7. Ensure that the **Compose the class visually** radio button is selected.
8. Click **Finish**. A new window appears with the **Visual Composition** tab selected.

Step 3: Add the Import Statements

You need to import a number of classes when running the executable file. Add the following import statements to the `Ex03MainDialog` class:

```
import com.ibm.connector.infrastructure.*;
import com.ibm.connector.infrastructure.java.*;
import com.ibm.connector.imstoc.*;
import com.ibm.connector.connectionmanager.*;
```

Step 4: Add Connection Manager and Run-Time Context

This step adds the run-time context and the connection manager, which is used for the GUI application.

Add the following statements to the `Ex03MainDialog` class:

```
ConnectionManager connMgr = new ConnectionManager();
JavaRuntimeContext runtimeContext = null;
```

Step 5: Create the Main GUI Dialog Interface

This step builds a GUI dialog for giving the input values to the command bean that was created, and for displaying the output values from the command bean. The GUI interface shown below is built using Swing components.

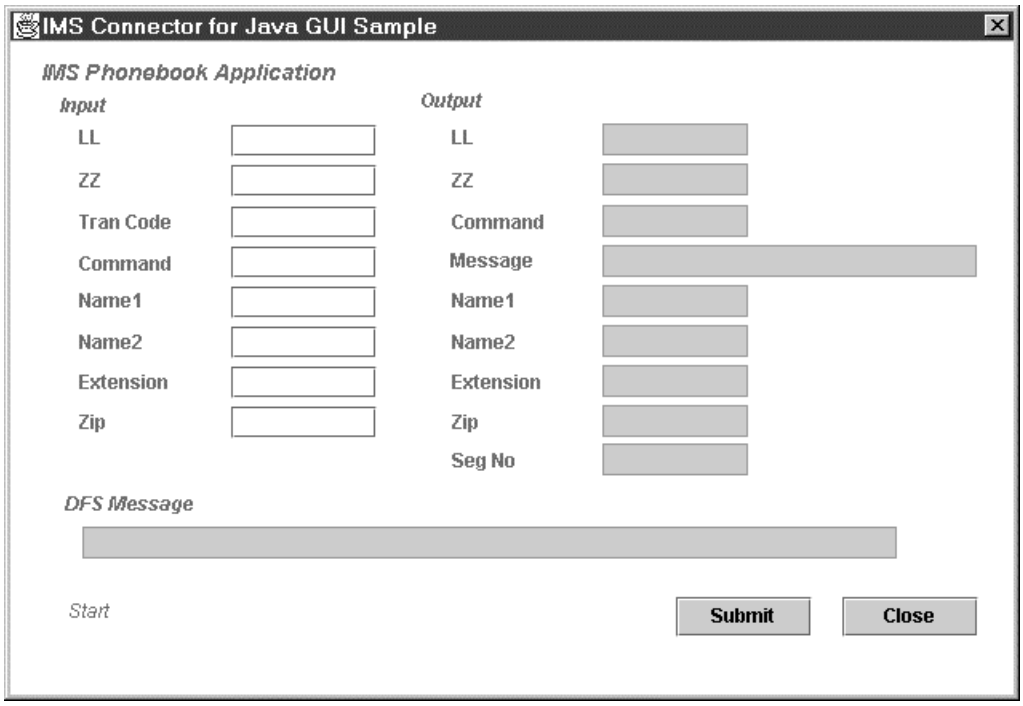


Figure 11. Sample GUI Dialog for the IMS Phone Book Application

Step 6: Create a Command to Execute the IMS Transaction

This step creates an EAB command to execute an IMS transaction, as described in Step 1: Create a Command Class for the IMS Transaction (page 40) under Creating

an EAB Command Using the Command Editor (page 39). Name this command class Ex03Command. When creating the Transaction Input/Ouput Record Beans for the command, make sure to do the following:

- On the **Create Record from Record Type** wizard, select the **Generate Notification** check box under **Additional Options**.

This ensures that appropriate notification is generated when the properties of the record bean are updated.

Step 7: Set Up the IMS Command

This step uses the Visual Composition Editor to associate the appropriate field values of IMS command created in the previous step, *Step 6: Create a Command to Execute the IMS Transaction*, with the GUI dialog to get input from the user.

1. In the Visual Composition Editor, click **Choose Bean...**, located beneath the **Swing** pull-down menu.
2. Click **Browse** to look through a list of available classes. In the **Pattern** field, enter `Ex03Command`. A list of class names appears.
3. Choose Ex03Command (in package ex03),for the Class name from the list of available classes. Click **OK** on this and the previous window.
4. To plant the bean, move the cursor to a position in an empty space in the Visual Composition Editor frame window and click the left mouse button. By default, it is assigned the name `Ex03Command1`.
5. The following connections need to be established:
   a. Set up the run-time context connection corresponding to this GUI thread. This step creates an Event-to-Code connection, such that the following block of code is executed when the actionPerformed event of the **Submit** button occurs (in other words, when the user clicks on the **Submit** button).
      - Click **Submit** and click the right mouse button. Choose **Connect —> actionPerformed**.
      - Move the cursor to an empty area in the frame window and click the left mouse button. Click **Event-to-Code...** and an **Event-to-Code Connection** dialog box appears.
      - In the **Event-to-Code connection** dialog box, replace the method with the code below. Replace the values `yourUID`, `yourPwd`, and `yourGrp` with the user ID, password, and group name for your IMS environment.

```
/**
  * Setup the runtime context and other property values to prepare
  * for the execution of an IMS transaction.  Provide the user ID,
  * password and group name that will be supplied to IMS.
  *
  */
 public void setup(Ex03Command cmd)
 {
   //-------------------------------+
   // Set up the runtime context for the current thread.  |
   //-------------------------------+
   if (runtimeContext == null)
   {
     runtimeContext = new JavaRuntimeContext();
     JavaRuntimeContext.setCurrent( runtimeContext );
   }
   //-------------------------------+
   // Provide your environment's values for user ID,       |
   // password, and group name for the runtime context.    |
```

```
        //-------------------------------+
        IMSLogonInfoItems logon =
          new IMSLogonInfoItems( runtimeContext.getLogonInfo(),
                                 cmd.getConnectionSpec() );
        logon.setUser( "yourUID" );
        logon.setPassword( "yourPwd" );
        logon.setGroup( "yourGrp" );
        //-------------------------------+
        // Set Connection Manager                                  |
        //-------------------------------+
        runtimeContext.setConnectionManager( connMgr );
        //-------------------------------+
        // Set trace level.                                        |
        //-------------------------------+
        (JavaRASService)runtimeContext.getRASService()).setTraceLevel(
          RASService.RAS_TRACE_OFF );
        return;
      }
```

- Click **OK**. Click **Yes** when prompted with **Text has been modified-Save changes?** dialog box. A new setup(Ex03Command cmd) method is created. A new connection is also created.
- This connection requires a `cmd` input parameter. To provide the `cmd` parameter and complete the connection, do the following:
  1) Select the new connection and click the right mouse button.
  2) Select **Connect -> cmd**.
  3) Move the cursor to the **Ex03Command1** bean icon and click the left mouse button. Click **this**.

b. Execute Command connection

This connection triggers the execution of the command bean when the submit button of the GUI is clicked:

- Select the new connection `setup(Ex03Command cmd)` created in the previous step and click the right mouse button.
- Select **normalResult**.
- Move the cursor to the **Ex03Command1** bean icon, and click the left mouse button.
- Click **Connectable Features...** and an **End connection** dialog box appears.
- Make sure that the **Method** radio button is selected. Scroll down the list to choose the execute() method and click **OK**. A new connection is created.

c. Input values property connections

This step associates the input properties of the command bean to the GUI interface to get input from the user.

This step creates property-to-property connections between the input text fields of the GUI and the input properties of the `Ex03Command1` bean.

- Select the **LL** text input field of the Input Section of the GUI and click the right mouse button. Select **Connect —> text**.
- Move the cursor to select the **Ex03Command1** icon and click the left mouse button. Click **Connectable Features...** An **End connection** dialog box appears.
- Scroll down the list of properties to choose the `IN__LL` and click **OK**.

- A property-to-property connection with the **LL** text input field on the GUI and the `IN__LL` property of the `Ex03Command1` bean is created.
- Select the newly created connection and click the right mouse button. Select **Properties**.
- In the source event box, select `caretUpdate`.
- Click **OK**. This keeps the value of the IN__LL property of the command updated when the input text is changed.
- Repeat the steps above to associate the following text field values of the input section of the GUI with the properties of the command bean:
  - ZZ —> IN__ZZ
  - Tran Code —> IN__TRCD
  - Command —> IN__CMD
  - Name1 —> IN__NAME1
  - Name2 —> IN__NAME2
  - Extension —> IN__EXTN
  - Zip —>IN__ZIP

  In a property-to-property connection, the ordering of the source and target bean affects how the property values are updated and synchronized.

d. Output values property connections

Similarly, this step associates the output property values of the command bean with the GUI interface to display the output to the user.

See the description in **Step 5, item d** on how to create property-to-property connections between the following output properties of the `Ex03Command1` bean and the output text fields in the **Output** section of the GUI:

- OUT__LL —> LL
- OUT__ZZ —> ZZ
- OUT__MSG —> Message
- OUT__CMD —> Command
- OUT__NAME1 —> Name1
- OUT__NAME2 —> Name2
- OUT__EXTN —> Extension
- OUT__ZIP —> Zip

In a property-to-property connection, the ordering of the source and target bean affects how the property values are updated and synchronized. In this step, ensure that the property-to-property connections are created with `Ex03Command1` as the source bean and the GUI output text fields as the target bean.

e. DFS message output connection

This connection associates the output `DFS` message from IMS to the GUI interface. The text value in the GUI for the `DFS` message is updated when the output bean of the command returned is a `DFSMsg` object.

1) Select the **Ex03Command1** icon and click the right mouse button. Choose **Connect —> Connectable Features...** A **Start connection** dialog box appears.
2) Make sure that the **Property** radio button is selected. Scroll down the list to choose the DFSDATA1 property and click **OK**.

3) Move the cursor to the **DFS Message TextField** in the GUI and click the left mouse button. Select **text**. A property-to-property connection with the **DFS Message Text** field and the DFSDATA1 property of the Ex03Command1 is created.

4) Select the newly created connection and click on the right mouse button. Select **Properties**.

5) In the **Source Event** box, select propertyChange. Click **OK**. This keeps the value of the **DFS Message Text** field updated when the DFSDATA1 property is changed (i.e., a DFS message is returned from IMS).

f. Close run-time context connection

This connection closes the run-time context when the **Close** button is clicked and the GUI closes.

This step creates an Event-to-Code connection, such that the following block of code is executed when the actionPerformed event of the **Close** button occurs (in other words, when the user clicks on the **Close** button):

```
/**
  * Close the runtime context for the current thread.
  */
 public void closeRuntimeContext()
 {
   //-------------------------------+
   // Commit resources and close the runtime context.            |
   //-------------------------------+
   if ( runtimeContext != null )
   {
     ((JavaCoordinator)runtimeContext.getCoordinator()).commit();
     runtimeContext.close();
     runtimeContext.removeCurrent();
   }
   //--------------------+
   // Dispose the window.                 |
   //--------------------+
   dispose();
   System.exit(0);
 }
```

See the description in **Step 5, item a** on how to create an Event-to-Code connection between the actionPerformed event of the **Close** button and the code. The code in this step does not requires any parameter.

Related Reading: For more information on property-to-property connections, see the *Visual Composition Editor* documentation.

Step 8: Finish

This is the final step in creating the GUI Application.

Click the **Bean** menu and click **Save Bean** to save your work. The workbench shows the new `Ex03MainDialog` class.

You can modify the handleException(Throwable) method of the `Ex03MainDialog` class to print messages when exceptions occur. All trace information and exceptions are logged to the console.

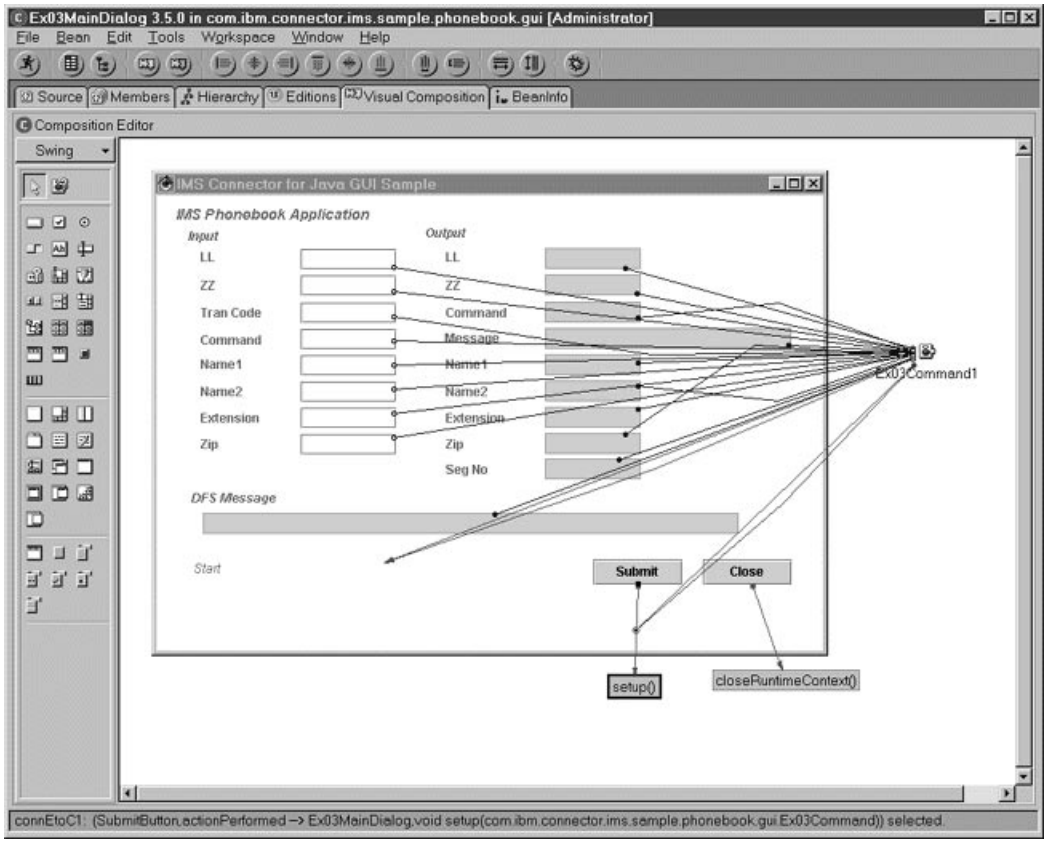


Figure 12. The Completed GUI Application

Chapter 10. Building a Java Servlet to Run an IMS Transaction

This section provides you with an example of how to use the IBM WebSphere Studio to create a Java servlet from your EAB command. This section also describes how to run your servlet in a WebSphere Application Server environment.

The process of creating and executing a servlet to run an IMS transaction is covered in the following sections:

1. Creating a Java Servlet Using WebSphere Studio
2. Running Your Java Servlets in a WebSphere Application Server Environment

Creating a Java Servlet Using WebSphere Studio

The IBM WebSphere Studio is a comprehensive tools environment for building Web applications. WebSphere Studio's Java Bean wizard can be used to create a Java servlet for the WebSphere Application Server from your IMS Connect command.

The steps include the following:

1. Configure WebSphere Studio
2. Provide Your EAB Command to WebSphere Studio
3. Start WebSphere Studio
4. Create a Project for Your Servlet
5. Start the Studio Wizard
6. Provide Input to the Studio Wizard
7. Modify the Code Generated by the Studio Wizard

Step 1: Configure WebSphere Studio

In order for WebSphere Studio to build a servlet from an Enterprise Access Builder command that uses IMS Connect, you must make the supporting class libraries available to WebSphere Studio.

Related Reading: For instructions on how to prepare your WebSphere Studio environment, see Preparing Your WebSphere Studio Environment.

Step 2: Provide Your EAB Command to WebSphere Studio

To use WebSphere Studio to build a servlet from your Enterprise Access Builder command, make the EAB command available to WebSphere Studio by copying its `.JAR` file to a target directory on your WebSphere Studio machine.

You can obtain this `.JAR` file by exporting it from VisualAge for Java as follows:

1. Highlight the package that contains the EAB command you created (for example, `ex01`).
2. Click **File —> Export**.
3. Click **Jar file** and click **Next**.
4. Name your `.jar` file in the text box at the top of the wizard. For example, ex01.jar.
5. Ensure that both the **class** and **beans** check boxes are checked.

© Copyright IBM Corp. 1998, 2000

6. Click the **Details...** button of the **beans** check box.
7. Ensure that the check box is marked on the **Projects** panel.
8. Ensure that **only** your EAB command is marked as a bean on the **Types** panel (for example, `Ex01Command`) and click **OK**.
9. Click **Finish** to export your `.jar` file.

Note:

WebSphere Studio integrates with VisualAge for Java, providing quick navigation between the two products. If you have VisualAge for Java installed and running on the same machine as your WebSphere Studio, you can easily transfer files like Java beans, servlets and files such as .java and .class between the two products.

If you are using the above feature to send your EAB Command bean (for example, ex01.Ex01Command shipped as the ex01\Ex01Command.class file) from VisualAge for Java, make sure that you also send all the related classes that will be used by your EAB command class (for example, the ex01.InMsg and ex01.OutMsg classes) in order to compile.

For instructions on how to use this feature, see **How To... —> Use Studio with VisualAge** in the WebSphere Studio 3.5's Studio Guide. See also **Concepts —> External tool integration —> Tool Integration API** in VisualAge for Java Online Help.

Step 3: Start WebSphere Studio

If you are using WebSphere Studio on Windows, click the **Start** icon and click **Programs —> IBM WebSphere —> Studio 3.5 —> IBM WebSphere Studio V3.5**. The WebSphere Studio window appears.

Step 4: Create a Project for Your Servlet

This step creates a project for your servlet.

If you do not yet have a WebSphere Studio project defined, select the **Create a new project using:** radio button, select **Default Template** from the scroll list and click **OK**, or from the **File** menu, click **New project**. The **New project** window appears. Enter a name for the new project file; for example, `ex01`. Choose a **Project Folder**, if desired. **Click OK**.

The steps that follow create a servlet that uses the Enterprise Access Builder command .jar file that was created in Step 2. This is the .jar file for the EAB command built. In addition to creating the servlet, WebSphere Studio produces an input HTML form for transaction input and to invoke the servlet and output `.jsp` files to display the results of running the transaction.

Note:

WebSphere Studio allows you to specify, for a particular Studio project, the version of WebSphere Application Server you will be using to run your servlet. Based on your selection, Studio varies how it handles processes such as code generation, compiling Java files, publishing, etc.

Related Reading: For instructions for setup of the Application Server preferences, see **How To... —> Manage your projects —> Specify server settings** in the WebSphere Studio 3.5's Studio Guide.

Step 5: Start the Studio Wizard

Select the **servlet folder** icon in the project tree (for example, `ex01`) and click **Insert —> File** to open the **Insert file** dialog. Click the **Use existing** tab to insert your Enterprise Access Builder command jar file that you created in Step 2: Provide Your EAB Command to WebSphere Studio.

Note: To obtain the EAB command and the related Java classes directly from VisualAge for Java using the integrated feature referred to in Step 2: Provide Your EAB Command to WebSphere Studio, do the following:

1. Select the **From External Source** tab of the **Insert File** dialog box.
2. Select **VisualAge for Java** under the list of **Providers**.
3. Click the **Browse** button and select the Java classes to be imported. (for example, select ex01.Ex01Command), and all its input/output beans, (for example, ex01.InMsg, etc.). Click **OK** to close the window. The related files (for example, ex01/Ex01Command.class, ex01/InMsg.class, etc...) will be shown under the list of **Files** of the **Insert File** dialog box.
4. Select the files from the **Files** list (for example, select all the .class files only) and click **OK** to finish the import.

Related Reading: For more information on using this integrated feature between WebSphere Studio and VisualAge for Java, see **How To...—>Use Studio with VisualAge** in the WebSphere Studio 3.5's Studio Guide. See also **Concepts —> External tool integration -> Tool Integration API** in VisualAge for Java's help.

To start the Studio wizard, select your project folder (for example, ex01), then select **Tools —> Wizards —> JavaBean Wizard** from the menu.

Step 6: Provide Input to the Studio Wizard

Enter the following on each of the Studio wizard pages:

Java Bean Wizard

Under the **Look in** scrollbox, select the Enterprise Access Builder command jar file (for example, `ex01.jar`) or the class file (for example, ex01/Ex01Command.class) that you have added to the project. Select the command bean name, qualifying it with the package name (for example, `ex01.Ex01Command`). Click **Next**. If you are using a .jar file and you don't see the .jar file, you may not have added the .jar file correctly in the previous step. Make sure that you have inserted the .jar file in the servlet folder of your project.

Markup Languages

Select the web application model and the markup language that will be used. Select the **Servlet Model** from the scroll list under **What style of code do you want to generate?**. Select **HTML** for the **Markup Language**.

Web Pages

Select the number of Web pages that you need. The **Create an input page** and **Create a results page** check boxes are selected as default. In our example, we also select the **Create a page for when an error occurs** check box.

Input Page

This form is used to enter the input data of your transaction. Check the fields that you want to appear on the input form. For example:

- hostName
- portNumber
- dataStoreName
- IN__CMD
- IN__EXTN
- IN__NAME1
- IN__NAME2
- IN__TRCD
- IN__ZIP

You can select a particular field and use the **Change** button to set initial values, or have more meaningful captions, or make other changes to the HTML. Click **Next**.

Because you typically do not present the **LL** and **ZZ** fields of an IMS transaction message to the end user, you should provide values for these fields by modifying the servlet code. See Step 7: Modify the Code Generated by the Studio Wizard, Item 2-d.

Results Page

This template is used to display the output data of your transaction. Check the fields that you want to appear on the output template. For example:

- OUT__CMD
- OUT__EXTN
- OUT__MSG
- OUT__NAME1
- OUT__NAME2
- OUT__ZIP
- OUT__SEGNO
- DFSDATA1

Only the first segment of the DFS message (shown as `DFSDATA1`) is displayed in this example. You can display all of the segments of the DFS message by promoting the **DFSDATA1..DFSDATA2** properties in your EAB command bean. Then, you can select them in this **Results Page** window of WebSphere Studio.

You can select a particular field and use the **Change...** button to change the captions. Click **Next**.

Standard Error Page

Select **Use this page** to display an existing error page on your Web server, or select **Create a new page with the following text** to let the wizard create one to be displayed when an error occurs. In this example, select the **Create a new page with the following text** radio button. Customize the error text message if desired. Click **Next**.

Methods

Select the **execute** method that has no parameters. Click **Next**.

Session

This bean can be used by more than one servlet in a Web application. You

can save this bean in the session and make it available, along with its data, for use on other Web pages within your Web site. For simplicity in this example, select **No**.

To ensure that each page has access to the correct EAB command bean, specify an alias name for the bean. For example, `ex01Command`. Click **Next**.

Finish The Servlet, Input, and Output pages are generated with the file names shown in the **Finish** window. Change the names files using the **Rename** button, if desired. (For this example, change the prefix to `Ex01Servlet`). Click **Finish** to save your files and return to WebSphere Studio.

The following files are generated from your EAB command:

- An input HTML form (for example, `Ex01ServletHTMLInput.html`)

This is the input HTML form that is used on a web browser to provide input data for your IMS transaction, and to invoke the servlet.

- An output `.jsp` page (for example, `Ex01ServletHTMLResults.jsp`)

This page is used on a web browser to display the output data of your IMS transaction.

- An error `.jsp` page (for example, `Ex01ServletHTMLError.jsp`)

This page is used on a web browser to display error information from your servlet.

- Java source for your servlet (for example, `Ex01Servlet.java`)
- A .servlet file for your servlet (for example, `Ex01Servlet.servlet`)
- A class file is created when you compile the Java source for your servlet. In addition, a `master.css` file is created as your style sheet.

Definitions:

- A *JSP file*, or *JavaServer Page*, is a type of file that WebSphere Studio generates to implement dynamic page content. JSP files can contain JavaScript coding, HTML tagging, and JSP tagging, and allow the WebSphere Application Server to dynamically add content to your HTML pages before they are sent to a requesting browser.

You can customize a JSP file to add your own text and images using JavaScript, HTML, or JSP tagging. This script will be included in the HTML file that is created by the WebSphere Application Server and returned to the requesting browser.

- A *.servlet file* contains information that is used to configure a servlet. You can modify a servlet's configuration by changing the entries in its .servlet file.

Step 7: Modify the Code Generated by the Studio Wizard

This step describes how you should modify the code that is generated by the Studio wizard. With the integration of WebSphere Studio and VisualAge for Java, you have the option of editing your files in VisualAge for Java, then sending the updated files back to the Studio. See *Step 2* above for more information.

Note: The modifications presented here represent a simple servlet and do not address issues relating to thread safety for a servlet.

To modify the code generated by the Studio Wizard, do the following:

Modify the Generated Java Source Code for Your Servlet

1. From the list of files in the WebSphere Studio window, select the file that contains the Java source for your servlet (for example, `Ex01Servlet.java`). From the **Tools** menu, click **Edit with** and select your editor (Notepad, for example) to open your file.

   If asked to check-out the associated file (for example, Ex01Servlet.class), select the check box of the associated file. Then, click on the Check Out and Edit button to continue.
2. Modify the performTask method as follows:

   a.
   ```
   Ensure that the following lines are commented out as shown:
   // com.ibm.connector.internal.LogonInfoItems logonInfoItems =
   // new com.ibm.connector.internal.LogonInfoItems(runtimeContext.getLogonInfo());
   // logonInfoItems.setUser("UserID");
   // logonInfoItems.setPassword("UserPassword");
   ```

   b. After the instantiation of the ex01 command bean, for example:
   ```
   ex01Command = (ex01.Ex01Command)java.beans.Beans.instantiate(getClass().getClassLoader(),
     "ex01.Ex01Command");
   ```

   add the following lines to set up the IMS LogonInfoItems class.
   ```
   com.ibm.connector.imstoc.IMSLogonInfoItems logon =
          new com.ibm.connector.imstoc.IMSLogonInfoItems(
            RuntimeContext.getCurrent().getLogonInfo(),ex01Command.getConnectionSpec());
   ```

   c. Set up the logon information for your transaction by either ″hard coding″ it directly into the Java source for your servlet, or by obtaining it from the input HTML form for the transaction. To ″hard code″ the information, add the following lines:
   ```
   logon.setUser("<UserName>");
   logon.setPassword("<Password>");
   logon.setGroup("<GroupName>");
   ```

   If you want to get the logon information from the input HTML form, modify the input HTML form that is generated by WebSphere (for example, `Ex01ServletHTMLInput.html`), as described in Modify the Input HTML Page for Your Servlet (Optional). As replacements to the lines above, add the following lines:
   ```
   logon.setUser(getParameter(request, "userName", true, true, true, null));
   logon.setPassword(getParameter(request, "password", true, true, true, null));
   logon.setGroup(getParameter(request, "group", true, true, true, null));
   ```

   d. Modify the generated code to setup the Common Connector Framework infrastructure services. For example,
   - Uncomment out the line
     ```
     runtimeContext.setConnectionManager (/* global instance of the class
      com.ibm.connector.connectionmanager.ConnectionManager */);
     ```

     to set your global instance of the Connection manager with the current runtime context.
   - Uncomment out the line
     ```
     ((JavaRASService)runtimeContext.getRASService()).setTraceLevel(RASService.RAS_TRACE_OFF);
     ```

     to setup your desired trace level of the RAS Service.

   See the comments in the generated code for more information.

   Related Readings: For more information about setting up the Common Connector Framework infrastructure, see **Task —> Accessing transaction**

systems —> Accessing CICS, IMS, Encina, MQSeries, HOD —> Setting up the Common Connector Framework RuntimeContext in VisualAge for Java Online Help.

e. Set the **LL** and **ZZ** fields prior to executing the transaction. The **LL** value is the length of all input message data fields, plus 4 (for LL and ZZ).

```
ex01Command.setIN__LL((short) <length>+4);
ex01Command.setIN__ZZ((short) 0);
```

3. From the list of files in the WebSphere Studio window, select the file that contains the Java source for your servlet (for example, `Ex01Servlet.java`).

   From the **Project** menu, click **Compile** to recompile your servlet after modifying the source.

   If the **Warning** dialog for replacing existing .class file appears, click **Yes**.

Modify the Input HTML Page for Your Servlet (Optional)

If you want to obtain the logon information from the input HTML form, do the following:

1. In the WebSphere Studio window, click on the project that contains the information for your servlet.
2. Select the file that contains the HTML source for your input page (for example, `Ex01ServletHTMLInput.html`) and open it for editing.
3. Add the following lines to the HTML input form so that the user can enter the user name, password, and group name:

```
<TR><TD>User Name</TD><TD><INPUT TYPE=TEXT NAME="userName" ID="userName">
   </TD></TR>
<TR><TD>Password</TD><TD><INPUT TYPE="password" NAME="password" ID="password">
   </TD></TR>
<TR><TD>Group</TD><TD><INPUT TYPE=TEXT NAME="group" ID="group">
   </TD></TR>
```

Modify the Output JSP Page for Your Servlet (Optional)

If you modify the output JSP file for your servlet, be sure that the bean name is entered correctly. The bean name is case-sensitive. Otherwise, an error is returned to the Web browser when WebSphere Application Server attempts to run your servlet.

Create a Web application on your WebSphere Application Server

WebSphere Application Server Version 3.5 supports the Web application programming model in which the servlets are now managed, configured and deployed based on the scope of a Web application (or Servlet group). A Web application is defined as a group of servlets that share the same servlet context.

Web applications are rooted in the directory path *<WAS_Install_Dir>*\hosts\default_host*<yourWebAppName>*, where *<WAS_Install_Dir>* is the installation directory of WebSphere Application Server and *<yourWebAppName>* is the name that identifies your Web application. For example, `d:\WebSphere\AppServer` (on Windows) or `/usr/WebSphere/AppServer` (on AIX).You must also configure the application classpath of your Web Application, which is <*WAS_install_dir*>\hosts\default_host\<yourwebappname>\servlets. All

compiled resources that a servlet of a Web application uses (for example, servlet .class files and the .jar files) should be placed in this directory. Resources in this directory are automatically reloaded whenever the resource is updated.

In addition, the non-Java resources such as .html and .jsp files will be placed in the Web application's document root instead of in the Web Server document root. The path of a Web application document root is <*WAS_install_dir*>\hosts\default_host\<*yourwebappname*>\web.

For more information on creating and configuring a Web application, refer to the *Configuring new Web applications* document of the WebSphere Application Server 3.5 documentation.

This step helps you build a Web application called *imsconn* that allows you to run your Java servlet.

1. Configure your Web application:
   a. Start the **Configure a Web application** task wizard by clicking **Console —> Tasks —> Configure a Web application** from the **WebSphere Administrative Console** menu bar.
   b. In the **Set Web App Name and Select System Servlets** pane, set the following properties:
      - Specify a name for the Web application in the Web App Name field. For example, *imsconn*.
      - Check the **Enable File Servlet** check box to enable the file servlet to serve HTML from the application server instead of the Web server.
      - Check the **Serve Servlets by Classname** check box to serve servlets directory from the Web application's servlets directory.
      - Specify the JavaServer Pages (JSP) specification to use. In this example, select the **Enable JSP 1.0** radio button.
   c. Select the **Next** button.
   d. Specify the servlet engine for your Web application. Click the + sign to expand **Nodes** tree. Click the + sign again on the node that has the TCP/IP host name of your Web server machine, to select the servlet engine under the server. For example, select **Default Servlet Engine**.
   e. Select the **Next** button.
   f. Name the Web application and the Web path for your application.
      1) Specify a name for your application in the **Web Application Name** field. For example, *imsconn*.
      2) Describe the Web application. (Optional).
      3) Specify the virtual host part of the Web application's served path in the **Virtual Host** field. For example, *default_host*.
      4) Specify the Web Path for the Web application in the **Web Application Web Path** field. For example, */webapp/imsconn*.
   g. Select the **Next** button.
   h. Specify the advanced setting.
      - Specify the document root for the Web application in the **Document Root** field. For example, <*websphere_install_dir*>\hosts\default_host\imsconn\web.
      - Specify the classpath for the Web application in the **Classpath for Application** field. For example, <*websphere_install_dir*>\hosts\default_host\imsconn\servlets.

- Specify other Web application properties or accept their default values.

i. Click the **Finished** button.

2. Restart your Web application:
   a. Go to the **Topology** tab by selecting **View —> Topology** from the menu bar.
   b. Expand the tree under the TCP/IP host name for your machine.
   c. Look for the imsconn Web application under **Default Server—>Default Servlet Engine.**
   d. Right click the imsconn Web application and choose **Restart Web App** from the popup menu.
3. Add folders representing your Web application (imsconn) to the WebSphere Application Server directory structure.
   a. Create the Web application folder. On Windows, create <*websphere_install_dir*>\hosts\default_host\imsconn. On AIX, create <*websphere_install_dir*>/hosts/default_host/imsconn.
   b. Create the Web application document root folder. On Windows, create <*websphere_install_dir*>\hosts\default_host\imsconn\web. On AIX, create <*websphere_install_dir*>/hosts/default_host/imsconn/web.
   c. Create the Web application classpath folder. On Windows, create <*websphere_install_dir*>\hosts\default_host\imsconn\servlets. On AIX, create <*websphere_install_dir*>/hosts/default_host/imsconn/servlets

Configuring the WebSphere Studio Publishing Service

WebSphere Studio provides a publishing service that enables you to easily move the generated HTML and .jsp pages and servlet class files to your Web server and WebSphere Application Server machine (local or remote). It also makes it easy to deploy your servlet as a Web application (as illustrated in this example). To set up the WebSphere Studio machine for the WebSphere Studio publishing services, do the following:

1. Start WebSphere Studio.
2. Choose a publishing stage whose files you want to publish by selecting **Project—>Publishing Stage —> your target assembly stage** (for example, `Test`).
3. Ensure that you are in **Publishing View**. To display the **Publishing View**, do one of the following:
   a. Click **View —> Publishing**.
   b. Click the **Publishing View** icon on the toolbar. The Publishing View is then shown in the right pane of WebSphere Studio.
4. You should see an icon for your target server(s) in the Publishing view. If there is no server shown, you need to add a Server in the Publishing view of your assembly stage. Do the following:
   a. Select your assembly stage (for example, Test) in the **Publishing View**.
   b. Click **Insert—>Server** from the menu.
   c. In the **Insert Server** dialog box, enter the TCP/IP hostname for your server (for example, *yourHost*) in the **Server name** field. An default server address would be provided (for example, *http://yourHost*), update the address in the **Server address** field if necessary.
   d. A server icon (for example, http://yourHost), will be added in the **Publishing View**.

5. Select the **Server** icon of your target server in the Publishing View; for example, `yourHost`. Click **Edit —> Properties** from the menu.
6. Select the method that you want to use to publish the files to your server. WebSphere Studio allows you to publish your files to the Web server using FTP or File System copy. For example, if you are publishing to a remote Web server, you might want to use FTP to move your files. If your Web server is a local machine, you can move your files using File system copy.

   To publish the files using FTP, do the following:

   a. In the **Publishing** tab of the **Properties** window of the server, select the **FTP publish** radio button.
   b. Provide the FTP logon information for the Web server environment.
   c. Provide the firewall information, if applicable.

   To publish the files using File System copy, do the following:

   a. In the **Publishing** tab of the **Properties** window of the server, select the **File system publish** radio button.
   b. Select the operating system on which the Application Server is running.
7. WebSphere Studio allows you to define publishing targets so that you can map your files in WebSphere Studio to specific directories on your Web server. The publishing targets specify directories on the Web server machine in which you want to place your individual files. To setup the publishing targets, do the following:

   a. Click on the **Targets** button.
   b. Define a publishing target for the web application's document root. This directory will contain the project's HTML, JSP and other resource files. By default, WebSphere Studio provides you with an **html** publishing target. All of the files in the root directory of your WebSphere Studio project are published to the Web server directory specified for the html publishing target. You must provide the appropriate directory path for this publishing target. For example, <*WAS_Install_Dir*>\hosts\default_host\imsconn\web.
   c. Define a publishing target for the web application's servlets and Java resources. This directory will contain the project's Java .class files and .servlet files and/or .jar files. By default, WebSphere Studio provides you with a **servlet** publishing target. All of the files in the servlet directory of your WebSphere Studio project are published to the Web server directory specified for the servlet publishing target. You must provide the appropriate directory path for this publishing target. For example, <*WAS_Install_Dir*>\hosts\default_host\imsconn\servlet.
   d. Click **OK**.
8. When publishing your servlet as a Web application, do the following:

   a. Specify the Web Application Web Path in the **Webapp web path** field in the **Publishing** tab of the publishing server's Properties window. For example, imsconn.
   b. Choose the publishing target you defined for the web application's document root directory. For example, html.
   c. Click **OK**.

Publishing to WebSphere Application Server 3.5

This step helps you publish your Web application to WebSphere Application Server Version 3.5 using WebSphere Studio 3.5.

Publish Your Java Servlet To Your Web Server

1. Make sure that all your files have been checked in to save all the latest changes. To check in a file, select the desired file and from the **Project** menu, select **Check In**.
2. You can publish the entire project or individual servers:
   - To publish the entire project, click the right mouse button on the desired WebSphere Studio project and select **Publish whole project**. Verify the setting in the **Publishing Options** window, and click **OK** to publish the files.
   - To publish a particular server in the project, do the following:
     a. Ensure that you are in the Publishing View. You can switch to the Publishing View by clicking **View —> Publishing** on the menu, or by clicking the **Publishing View** icon on the toolbar. The Publishing View is shown in the right pane of WebSphere Studio.
     b. Click the right mouse button on the **Server** icon of your target server and select **Publish this Server**.
     c. Verify the setting in the **Publishing Options** window. Click **OK** to publish the files.
3. If you are using FTP as your publishing mechanism, enter the port, login, and password information and click **OK**.

Running Your Java Servlets in a WebSphere Application Server Environment

This section describes how to run a Java servlet using the WebSphere Application Server. To run a Java servlet, do the following:

1. Start WebSphere Application Server 3.5 (you must first start the Web server, and then start WebSphere Application Server separately)

   Using Windows

   a. To start your Web server, do the following, for example, on an IBM HTTP server:
      1) Click **Start—>Settings—>Control Panel**. Click the **Services** icon.
      2) Select **IBM HTTP Server** and click **Start**.
      3) Click **Close** to close the window.

   b.

      To start the WebSphere Application Server 3.5, do the following:
      1) Click **Start—>Settings—>Control Panel**. Click the **Services** icon. Select **IBM WS AdminServer** and click **Start**.
      2) Click **Close** to close the window.
      3) Click **Start—>Programs—>IBM WebSphere—>Application Server V3.5—>Administrator's Console**.
      4) Ensure that the **Topology** view is displayed.
      5) To switch to the **Topology** view, choose **View—>Topology** from the menu bar. Click the + sign to expand the view of the **WebSphere Administrative Domain** node.
      6) Click on the + sign again to expand the view of the node that has the TCP/IP host name of your Web server machine.

7) Right click on **Default Server** under the host node in the previous step, and select **Start**. A dialog box displaying ″Command 'DefaultServer.start' completed successfully″ will appear when the server has been started. Click **OK** on the dialog box.

Using AIX

a.

To start your Web server, type one of the following on the AIX command line:

1) For a Domino Go server, enter: `startsrc -s httpd`
2) For an IBM HTTP server, enter: `apachectl start`

b.

To start WebSphere Application Server 3.5, do the following:

1) Click **Start—>Settings—>Control Panel**. Click the **Services** icon. Select **IBM WS AdminServer** and click **Start**.
2) Click **Close** to close the window.
3) Click **Start—>Programs—>IBM WebSphere—>Application Server V3.5—>Administrator's Console**.
4) Ensure that the **Topology** view is displayed.
5) To switch to the **Topology** view, choose **View—>Topology** from the menu bar. Click the + sign to expand the view of the **WebSphere Administrative Domain** node.
6) Click on the + sign again to expand the view of the node that has the TCP/IP host name of your Web server machine.
7) Right click on **Default Server** under the host node in the previous step, and select **Start**. A dialog box displaying ″Command 'DefaultServer.start' completed successfully″ will appear when the server has been started. Click **OK** on the dialog box.

2. Run your Java servlet on WebSphere Application Server 3.5:
   a. From a Web browser, type the following URL and press enter:
      1)

      `http://<servername>/webapp/imsconn/yourServletHTMLInput.html`where <servername> is the name of your Web server (for example, `Server1.stl.ibm.com`) and `yourServletHTMLInput.html` is the name of the HTML that is generated by WebSphere Studio (for example, `Ex01ServletHTMLInput.html`).
      2) Enter the required input data for the IMS transaction.
      3) When the **Submit** button on the input form is clicked, the Java servlet is invoked, the IMS transaction is run, and the IMS transaction output data is returned to the Web browser.

Chapter 11. Building a Web Application that Uses One Servlet to Run an IMS Conversation

This section describes how to build a Web application with one servlet that uses IMS Connector for Java to run an IMS conversation. It includes descriptions of how to create record beans that represent the input and output messages of the iterations of the IMS conversation, how to build an EAB command for the conversation using the VisualAge for Java Command Editor, how to use WebSphere Studio to create a template for the Web application, then finally how to build the complete Web application and use it to run the IMS conversation. These steps are similar to the steps for building a nonconversational application.

This example uses the conversational IMS transaction that is described in the IMS INSTALL/IVP Sample Application.

Two of the packages in the IMS Connector Samples feature pertain to this example. Java source similar to the Java source for the EAB command of this example can be found in the **com.ibm.connector.ims.sample.conv.phonebook** package of the Connector IMS Samples project. Java source similar to the Java source of the servlet for this example is contained in the **com.ibm.connector.ims.sample.conv.phonebook.servlet** package of the Connector IMS Samples project. These packages are added to your workspace when the IMS Connector Samples feature is added. See Preparing your VisualAge for Java Environment for instructions on installing the IMS Connector Samples feature if you have not already done so.

In addition, other Web application files such as the HTML and JSP files, JAR file, the SERVLET and CLASS files in their appropriate folder(s), and a theme folder containing Master.css can be found in *<IBM_Connectors_install_dir>*\imsconn\samples\servlet\convphonebook, where *<IBM_Connectors_install_dir>* is the IBM Connectors installation directory on the drive on which VisualAge for Java is installed.

By following the instructions in this section, you will create an application which is the same as the corresponding sample, except that it uses different project and package names than are used in the sample since you can't have duplicate package or project names within your VisualAge environment. As a result, you will be able to check your work against the sample, helping you to resolve any difficulties you encounter as you progress through the steps that follow.

Creating Input and Output Record Beans for an IMS Conversation

An IMS conversation is made up of a series of connected interactions between a client and, in the simplest case, a single IMS application program (transaction). More complex IMS conversations can be made up of multiple IMS application programs (transactions). Each interaction in the conversation is referred to as an *iteration* of the conversation. The input and output messages for the iterations of a conversation are often different.

Java record beans for an IMS conversation are created by VisualAge for Java from the COBOL source file(s) that contain the data structures (01 commareas) for the input and output messages of each iteration of the IMS conversation. These file(s)

© Copyright IBM Corp. 1998, 2000

are provided to VisualAge for Java's COBOL importer. For this example, data structures for all the iterations of the conversation are found in the single IMS application program, dfsiva34.mem, which can be found in <*IBM_Connectors_install_dir*>\imsconn\samples\misc, where <*IBM_Connectors_install_dir*> is the IBM Connectors installation directory.

For this particular IMS conversation, the iterations of the conversation consist of the following:

- The first iteration of the conversation invokes the IMS application program with an input message, which will be referred to as MI1, and returns an output message, which will be referred to as MO2. The input message, MI1, is represented by the COBOL 01 data structure INPUT-MSG. As is the case for all IMS conversational application programs, INPUT-MSG does not map the transaction code of the input message. The output message, MO2, is represented by the COBOL 01 data structure OUTPUT-AREA.
- The second and subsequent (middle) iterations of the conversation invoke the IMS application program with an input message, which will be referred to as MI2, and return the same output message as the first iteration, MO2. The input message, MI2, is completely represented by the COBOL 01 data structure INPUT-MSG. The output message, MO2, is represented by the COBOL 01 data structure OUTPUT-AREA.
- The conversation is terminated by the IMS application program on receipt of an ″END command″ input message. This message is sent in response to the end-user selecting the ″END command″ radio button. The servlet detects that the IMS application program has ended the conversation using the getConvEnded() method of the IMSInteractionSpec class. The servlet then displays the final JSP page of the conversation.

Note: The process of understanding the input and output messages of the iterations of an IMS conversation is key to developing your Web application. If the IMS transaction has MFS source, the MSG statements, along with the NXT= keywords, can sometimes aid in this process.

The steps to build record beans representing the input and output messages presented above include the following:

1. Start the Workbench in the VisualAge for Java IDE
2. Create a Project and Package for the IMS Conversational Input and Output Record Beans and EAB Command
3. Create a COBOL RecordType Class for the Initial Input Message of the IMS Conversation
4. Create a Transaction Input Record Bean for the Initial Input Message
5. Create a COBOL RecordType Class for the Second Input Message
6. Create a Record Bean for the Second Input Message
7. Create a COBOL RecordType Class for the Output Message
8. Create a Record Bean for the Output Message

Step 1: Start the Workbench in the VisualAge for Java IDE

For a description of how to start the Workbench in the VisualAge for Java IDE, see Step 1: Start the Workbench in the VisualAge for Java IDE (page 36) under Creating IMS Transaction Input and Output Beans (page 36) in Building a Java Application to Run an IMS Transaction.

Step 2: Create a Project and Package for the IMS Conversational Input and Output Record Beans and EAB Command

For a description of this step, see Step 2: Create a Project and Package for the Application (page 36) under Creating IMS Transaction Input and Output Beans (page 36) in Building a Java Application to Run an IMS Transaction. Use Examples for the project name and convcmds for the package name.

Step 3: Create a COBOL RecordType Class for the Initial Input Message of the IMS Conversation

This step parses the COBOL commarea that represents the initial input message for the IMS conversation, and creates a COBOL RecordType class that encapsulates the information in the commarea. This class will be used in *Step 4: Create an Input Record Bean for the Initial Input Message* below to build a Java bean that represents the input message. Do the following:

1. Ensure that the new package, convcmds, is selected. From the **Selected** menu, click **Tools —> Enterprise Access Builder —> Import COBOL to RecordType...** The **Import COBOL to RecordType** wizard appears.
2. On the **Import COBOL to RecordType** wizard, do the following:
   - Enter the path and file name of the COBOL file that contains the commarea representing the IMS transaction input message (dfsiva34.mem), select the **An IMS Application Program** radio button, then click **Next**. The next window of the **Import COBOL to RecordType** wizard appears, and contains a list of the available level 01 commareas.
3. Select the commarea representing the IMS transaction input message from the **Available** level 01 commarea list (for example, INPUT-MSG)) and add it to the **Selected** commareas list by clicking the **>** button.
4. Select the **Use BigDecimal** check box. This will generate the COBOL type as a BigDecimal Java type.
5. Add a transaction code field to the COBOL RecordType.

   During the first iteration of a conversation, IMS Transaction Manager (IMS TM) removes the transaction code from the input message and places it in a scratch pad area (SPA) accessible to the IMS application program. The SPA is also used by the IMS application program to save information between iterations of the conversation.

   The IMS application program gets the contents of the SPA with a GU call, then gets the data portion of the initial input message with a GN call. The IMS application program does not see the transaction code in the I/O area that it gets using the GN call, since the transaction code has already been removed from the initial input message by IMS TM. Hence, in an IMS conversational application program, there is no 01 data structure that directly maps to the input message of the first iteration. For this reason, you must indicate to the Enterprise Access Builder tool the information it needs to build a COBOL RecordType class, and subsequently a record bean, representing the complete initial input message.

   Use the following steps to complete the initial input message by adding a transaction code field:

   a. Select the **Add transaction code field** check box.

   b. Enter the name you would like to use for the transaction code in the **Field name** field (for example, IN__TRANCD).

      This is the name of the field in the generated Java record bean, and hence the name used in the corresponding ″get″ and ″set″ methods. In this case,

IN__TRANCD was selected to be consistent with the names of the ″get″ and ″set″ methods that were generated for the fields in the COBOL 01 data structure, INPUT-MSG).

c. Select the length of the transaction code for the **Length** field (for example, 5).

The Enterprise Access Builder tool builds the record bean for the initial input message by placing the transaction code immediately following the ZZ field. A transaction code has a maximum length of 8 (EBCDIC) single-byte characters. If the length of the transaction code is less than 8 characters, the Enterprise Access Builder tool separates the transaction code from the input message data by a single (EBCDIC) blank character. For example, if the transaction code is ″IVTCB″, you specify 5 for the **Length** field and the tool creates a record bean of the form:

*LLZZ***IVTCBb***Data*..., where

- **b** represents a single blank (EBCDIC) character, and
- *Data*... represents the portion of the initial input message represented by the 01 COBOL data structure (e.g., INPUT-MSG), following the definitions for the LL and ZZ fields.

If the length of the transaction code is equal to 8 characters, the Enterprise Access Builder tool *does not* separate the transaction code from the input message data. For example, if the transaction code is ″CARLOANS″, you specify 8 for the **Length** field and the tool creates a record bean of the form:

*LLZZ***CARLOANS***Data*...

d. Enter the value of the transaction code in the **Value** field (for example, IVTCB).

This is optional, since you may choose to provide the value of the transaction code at runtime, as is done in this example. Click **Next**.

6. Ensure that the correct names appear in the **Project Name** and **Package** entry fields.
7. Enter a name for the new class that will represent the COBOL RecordType. For example, MI1Type.
8. To continue to generate the record bean from this record type, ensure that both the **Continue working with newly created record type** check box and the **Create record from record type** radio button are selected.
9. Click **Finish**. A new class named MI1Type appears in the package named convcmds in the IDE Workbench.

The following fragment of COBOL source code is the 01 commarea used to generate the COBOL RecordType class for the initial input message of the conversation:

```
01  INPUT-MSG.
         02  IN-LL          PICTURE S9(3) COMP.
         02  IN-ZZ          PICTURE S9(3) COMP.
         02  IN-FILL        PICTURE X(4).
         02  IN-COMMAND     PICTURE X(8).
         02  TEMP-COMMAND REDEFINES IN-COMMAND.
             04  TEMP-IOCMD    PIC X(3).
             04  TEMP-FILLER   PIC X(5).
         02  IN-LAST-NAME   PICTURE X(10).
         02  IN-FIRST-NAME  PICTURE X(10).
         02  IN-EXTENSION   PICTURE X(10).
         02  IN-ZIP-CODE    PICTURE X(7).
```

Step 4: Create an Input Record Bean for the Initial Input Message

This step creates a record bean that represents the initial input message of the IMS conversation from the COBOL RecordType class created in *Step 3: Create a COBOL RecordType Class for the Initial Input Message of the IMS Conversation* above. Do the following:

1. If both the **Continue working with newly created record type** check box and the **Create record from record type** radio button are selected in *Step 3: Create a COBOL RecordType Class for the Initial Input Message of the IMS Conversation*, the **Create Record from Record Type** wizard appears. To manually start the **Create Record from Record Type** wizard, ensure that the COBOL RecordType class, MI1Type, is selected. From the **Selected** menu, click **Tools —> Enterprise Access Builder —> Create Record from Record Type...** The **Create Record from Record Type** wizard appears.
2. On the **Create Record from Record Type** wizard, enter the following:
   - Ensure that the correct names appear in the **Project Name** and **Package** entry fields.
   - Enter a name for the new class that will represent the input message. For example, MI1.
   - Click **Access Method: Direct**.
   - Click **Record Style: Custom Records**.
   - In this example, under **Additional Options**, **Generate with Notification**, **Use Inner Classes**, and **Shorten Names** are *not* checked. Although it is sometimes necessary or desirable to use one or more of these options when creating record beans, these options are not selected in this example. Not selecting **Generate with Notification** and **Use Inner Classes** provides better performance.
3. Click **Next** to view the next **Create Record from Record Type** wizard. Ensure that this wizard shows the following values, which indicate that the IMS transaction input message is processed on an MVS host machine:
   - Floating Point Format is IBM.
   - Endian is Big Endian.
   - Remote Integer Endian is Big Endian.
   - Code Page is 037.
   - Machine Type is MVS.
4. Click **Finish** on the Create Record from Record Type wizard. Two new classes, called MI1BeanInfo and MI1, appear in the package convcmds in the IDE Workbench. The MI1 Java bean represents the initial input message of the IMS conversation.

Notes:

Code page refers to the data of the IMS transaction. If your transaction data is other than U.S. English (code page 037), you must enter a different code page in this field.

Step 5: Create a COBOL RecordType Class for the Second Input Message

This step creates a COBOL RecordType for the input message of the second and subsequent iterations of the conversation. It is created from the 01 COBOL data structure, INPUT-MSG. For a description of how to create a COBOL RecordType

class from the COBOL source for an IMS input message, see Step 3: Create a COBOL RecordType Class for the IMS Transaction Input Message under Creating IMS Transaction Input and Output Beans in Building a Java Application to Run an IMS Transaction. ***** reference goes here *****. For this example, specify MI2Type for the name of the COBOL RecordType class.

In this case, the input message (MI2) is completely represented by the 01 COBOL data structure, INPUT-MSG. On the **Import COBOL to RecordType** wizard, you can select either the **An IMS Application Program** radio button or the **Generic COBOL code** radio button and you *do not* select the **Add transaction code field** check box.

The following fragment of COBOL source code is the 01 commarea used to generate the COBOL RecordType class for the second input message of the conversation:

```
01  INPUT-MSG.
          02  IN-LL          PICTURE S9(3) COMP.
          02  IN-ZZ          PICTURE S9(3) COMP.
          02  IN-FILL        PICTURE X(4).
          02  IN-COMMAND     PICTURE X(8).
          02  TEMP-COMMAND REDEFINES IN-COMMAND.
              04  TEMP-IOCMD    PIC X(3).
              04  TEMP-FILLER   PIC X(5).
          02  IN-LAST-NAME   PICTURE X(10).
          02  IN-FIRST-NAME  PICTURE X(10).
          02  IN-EXTENSION   PICTURE X(10).
          02  IN-ZIP-CODE    PICTURE X(7).
```

Step 6: Create a Record Bean for the Second Input Message

This step creates a Record Bean representing the input message of the second and subsequent iterations of the conversation. For a description of how to create a record bean from the COBOL RecordType class, see Step 4: Create a Transaction Input Record Bean (page 38) under Creating IMS Transaction Input and Output Beans (page 36) in Building a Java Application to Run an IMS Transaction. For this example, specify MI2 for the name of the record bean. An additional class MI2BeanInfo will also be created.

Step 7: Create a COBOL RecordType Class for the Output Message

This step creates a COBOL RecordType for the output message of the first, second, (and subsequent) iterations of the conversation. It is created from the 01 COBOL data structure, OUTPUT-AREA. For a description of how to create a COBOL RecordType class from the COBOL source for an IMS output message, see Step 5: Create a COBOL RecordType Class for the IMS Transaction Output Message (page 39) under Creating IMS Transaction Input and Output Beans (page 36) in Building a Java Application to Run an IMS Transaction. For this example, specify MO2Type for the name of the COBOL RecordType class.

In this case, on the **Import COBOL to RecordType** wizard, you can select either the **An IMS Application Program** radio button or the **Generic COBOL code** radio button, but *do not* select the **Add transaction code field** check box.

The following fragment of COBOL source code is the 01 commarea used to generate the COBOL RecordType class for the output message of the conversation:

```
01  OUTPUT-AREA.
         02  OUT-LL       PICTURE S9(3) COMP VALUE +95.
         02  OUT-ZZ       PICTURE S9(3) COMP VALUE +0.
         02  OUTPUT-LINE  PICTURE X(85) VALUE SPACES.
         02  OUTPUT-DATA REDEFINES OUTPUT-LINE.
             04  OUT-MESSAGE   PIC X(40).
             04  OUT-COMMAND   PIC X(8).
             04  OUT-DATA-TYPE.
                 06  OUT-LAST-NAME   PIC X(10).
                 06  OUT-FIRST-NAME  PIC X(10).
                 06  OUT-EXTENSION   PIC X(10).
                 06  OUT-ZIP-CODE    PIC X(7).
         02  OUT-SEGMENT-NO  PICTURE X(4) VALUE '0001'.
         02  OUT-FILL        PICTURE X(2).
```

Step 8: Create a Record Bean for the Output Message

This step creates a Record Bean representing the output message of the first, second, (and subsequent) iterations of the conversation. For a description of how to create a record bean from the COBOL RecordType class, see Step 6: Create a Transaction Output Record Bean (page 39) under Creating IMS Transaction Input and Output Beans (page 36) in Building a Java Application to Run an IMS Transaction. For this example, specify MO2 for the name of the record bean. An additional class MO2BeanInfo will also be created.

Building an EAB Command for the IMS Conversation

This section describes how you create the Enterprise Access Builder (EAB) command for this example's IMS conversation using the VisualAge for Java Command Editor. The Command Editor guides you through the steps of constructing a command, allowing you to focus on the specific composition patterns of a command.

The Web application for this example is modeled using a single EAB command. For IMS conversations, which consist of multiple interactions with an IMS application program, there may be multiple ways in which you could model your Web application. Possible models are:

- A single EAB command is used to represent the entire conversation.

A single EAB command bean can be used for all the iterations of an IMS conversation. Input and output record beans are created to represent all possible input and output messages for all possible iterations of the conversation. Two approaches for associating these beans with the single EAB command are presented below. The approach that you choose to use may vary depending on the nature of your IMS conversation. In both cases the Java servlet determines the appropriate input bean to be used for a particular iteration of the conversation at runtime. It uses the setInput() method of the EAB command to dynamically associate the input record bean with the EAB command at runtime.

1. Output record beans are associated with the EAB command at development time. The EAB command includes output beans for all possible iterations of the conversation. After the input record bean has been associated with the EAB command at runtime using the setInput() method, the execute() method of the EAB command is called to interact with the IMS conversational application program, and one of the output record beans provided at development time is returned. These steps are repeated by the servlet for the duration of the conversation .

2. The output of the EAB command is represented as a single class that returns a byte array. Output record beans, separate from the EAB command, are created for the all the possible iterations of the conversation. After the input record bean has been associated with the EAB command at runtime using the setInput() method, the execute() method of the EAB command is called to interact with the IMS conversational application program, and the output of the iteration is returned in the byte array. The servlet determines which of the output record beans to populate with the bytes returned in the array. This determination could be made using, for example, the value of the mapname returned in the IMSInteractionSpec of the EAB command. These steps are repeated by the servlet for the duration of the conversation.

- Multiple EAB commands are used, each representing a separate iteration of the conversation.

In this scenario, a different EAB command is invoked for each iteration of the conversation. Each EAB command has input and output record beans representing the input and output messages of the particular iteration it represents. The input and output record beans are set during development of the EAB command.

- A combination of the above.

The steps for creating the EAB command for this example are presented below. The EAB command is created in the same project and package as the input and output record beans described above.

1. Create an EAB Command Class for the IMS Conversation
2. Promote the Connection Specification Properties
3. Promote the Interaction Specification Properties
4. Promote the Output Record Bean Properties
5. Create the Command Bean

Step 1: Create an EAB Command Class for the IMS Conversation

For a description of how to create a command class, see Step 1: Create a Command Class for the IMS Transaction (page 40) under Creating an EAB Command Using the Command Editor (page 39) in Building a Java Application to Run an IMS Transaction. For this particular example, the EAB command should be created as follows:

- Use ConversationalCommand for the name of the EAB command class
- Add com.ibm.connector.imstoc.IMSConnectionSpec to the EAB command class
- Add com.ibm.connector.imstoc.IMSInteractionSpec to the EAB command class
- Add output bean convcmds.MO2 to the EAB command class
- Add output bean com.ibm.connector.imstoc.DFSMsg to the EAB command class
- **Do not** add an input bean to the EAB command class

Step 2: Promote the Connection Specification Properties

For a description of how to promote connection specification properties, see Step 6: Promote the Connection Specification Properties (page 45) under Creating an EAB Command Using the Command Editor (page 39) in Building a Java Application to Run an IMS Transaction. For this particular example, promote the following properties:

- Port
- Host name

Step 3: Promote the Interaction Specification Properties

For a description of how to promote interaction specification properties, see Step 5: Promote the Interaction Specification Properties (page 44) under Creating an EAB Command Using the Command Editor (page 39) in Building a Java Application to Run an IMS Transaction. For this particular example, promote the following properties:

- Datastore name
- Conversation ended

Step 4: Promote the Output Record Bean Properties

For a description of how to promote output message bean properties, see Step 3: Promote the IMS Transaction Output Properties (page 43) under Creating an EAB Command Using the Command Editor (page 39) in Building a Java Application to Run an IMS Transaction. For this particular example, promote the following properties:

- OUT__FIRST__NAME
- OUT__MESSAGE
- OUT__LAST__NAME
- OUT__EXTENSION
- OUT__ZIP__CODE

Step 5: Create the Command Bean

For a description of how to create the command bean, see Step 7: Create the Command Bean (page 45) under Creating an EAB Command Using the Command Editor (page 39) in Building a Java Application to Run an IMS Transaction.

Note: For this example, the getDFSMessage() method of com.ibm.connector.imstoc.DFSMsg is added to the EAB command class (ConversationalCommand). For a description of adding this method see Step 4: Promote the DFS Message Properties (page 43) under Creating an EAB Command Using the Command Editor (page 39) in Building a Java Application to Run an IMS Transaction.

Creating a Conversational Web application

After the EAB command bean has been built, it is exported in the form of a JAR file. The JAR file is used by WebSphere Studio to generate a template for the Web application, and at runtime for access to the classes of the EAB command. WebSphere Studio generates a Web application consisting of a single servlet, a single input HTML page, a single output JSP page, and an error page. The Web application built by WebSphere Studio is used as a template from which to build the complete conversational Web application. The Web application needs to be extended significantly in order to comply with IMS Connector for Java's Conversational HttpSession programming model. This example, as well as the sample in the **com.ibm.connector.ims.sample.conv.phonebook.servlet** package of the Connector IMS Samples project, are built using WebSphere Studio 3.0.2. However, a similar procedure is used for later releases of WebSphere Studio.

The following sections summarize the steps involved in developing the conversational Web application. The reader may wish to examine the files of the corresponding sample while developing their own Web application.

Step 1: Use WebSphere Studio to Create a Web Application Template

This step is summarized in the following steps. For a detailed description of these steps see Building a Java Servlet to Run an IMS Transaction.

1. Use VisualAge for Java to create a JAR file for the EAB command. For a description of how to create a JAR file for an EAB command, see Step 2: Provide Your EAB Command to WebSphere Studio (page 95) in Creating a Java Servlet Using WebSphere Studio (page 95) under Building a Java Servlet to Run an IMS Transaction. When creating the JAR file, mark the class ConversationalCommand as a bean.
2. In WebSphere Studio, create a project for your Web application, for example convphoneserv.
3. Insert the above JAR file in the servlet folder of your project and start the **JavaBean Wizard**.
4. In the **JavaBean Wizard**, choose to create an input page, a results page, and an error page.
5. On the Input Page of the **JavaBean Wizard** select fields *hostName*, *portNumber*, and *dataStoreName*. Additional fields will be added to the input HTML page in a later step, as well as more appropriate captions.
6. On the Results Page of the **JavaBean Wizard** select fields *OUT__EXTENSION*, *OUT__FIRST__NAME*, *OUT__LAST__NAME*, *OUT__MESSAGE*, and *OUT__ZIP*. Additional fields will be added to the output JSP page in a later step, as well as more appropriate captions.
7. On the Methods page of the **JavaBean Wizard** select the *execute()* method.
8. On the Session page of the **JavaBean Wizard**, in answer to *Will you use this bean or query on more than one page?*, select the *Yes, store it in the user's session()* radio button.
9. On the Finish page of the **JavaBean Wizard** select the *Rename...* button and choose ConvIterations as the prefix for the generated file names.

The following files are generated by WebSphere Studio and are used as a template for the Web application:

- ConvIterationsInput.html
- ConvIterations.java
- ConvIterationsResults.jsp
- ConvIterationsError.jsp
- ConvIterations.servlet
- Master.css, in a folder called *Theme*

The above Web application template must be modified extensively in order to create a conversational Web application. Files must be modified and new JSP pages must be added. The modified conversational Web application will consist of the following:

- ConvIterationsInput.html

  This page will be modified to allow the user to provide the data for the input message of the first iteration of the conversation, MI1.
- ConvIterationsResults.jsp

  This page will be modified to display the output message of the first and subsequent iterations of the conversation. In addition, it will be modified to allow the user to provide the data for the input message of subsequent iterations of the conversation, MI2.
- ConvIterations.java

The Java source for the servlet will be modified, to implement the Conversational HttpSession programming model, to be invoked by two types of request objects, and to display a page with the results of an iteration (ConvIterationsResults.jsp), a DFS message page (ConvDFSMessage.jsp), or a page for the end of the conversation (ConvEndConversation.jsp).

- ConvDFSMessage.jsp

  This page will be added to the Web application. It will display any ″DFS″ messages returned by IMS while the IMS conversation is running.

- ConvEndConversation.jsp

  This page will be added to the Web application. It will display a separate page indicating that the IMS conversation has ended.

- ConvIterationsError.jsp

  This is the error page generated by WebSphere Studio. No changes are made to this file.

- ConvIterations.servlet

  This file, the servlet configuration file, will be modified to allow the callPageNamed() method to be used to display the JSP pages described above.

- Master.css, in a folder called *Theme*

  This is the style sheet generated by WebSphere Studio. No changes are made to this file.

Step 2: Modify the Initial Input HTML Page, ConvIterationsInput.html

The basic structure of this page remains the same. Input fields are added for the fields of the initial input message of the IMS conversation, MI1. These fields were not available in WebSphere Studio, since the input message is associated with the EAB command at runtime using the setInput() method. The fields added are IN__FIRST__NAME, IN__LAST__NAME, IN__EXTENSION, IN__ZIP__CODE, and IN__COMMAND. Radio buttons are used to provide a value for the IN__COMMAND field.

Template fields hostName, portNumber, and dataStoreName are kept, but new captions and positioning are provided. In addition, fields userID, password, and group are added. These 6 fields will only be provided for input on the initial HTML page.

A hidden field is used to indicate which page type invokes the conversational servlet. For ConvIterationsInput.html the hidden field is indicated as follows:

```
<input type="hidden" name="pageName" value="StartPage">
```

See the file ConvIterationsInput.html in <*IBM_Connectors_install_dir*>\imsconn\samples\servlet\convphonebook for an illustration of the modifications described in this step.

Step 3: Modify the Results JSP Page, ConvIterationsResults.jsp

The basic structure of this page must be changed from an output JSP page to a combination of an output JSP page, to display the data of the output message of an iteration (MO2), and a data input form, to accept the data for the input message of the next iteration (MI2). The changes are as follows:

- Change the page into a data input form by adding a <FORM></FORM> element and **Submit** and **Reset** buttons. The <FORM> statement should invoke the conversational servlet as follows:

```
<FORM METHOD="post" ACTION="/servlet/convphoneserv.ConvIterations">
```

- Change the output fields to input/output fields by adding the <INPUT> tag. The value of the IN__COMMAND field is input via radio buttons in the same manner as for the initial input HTML page.
- Add a hidden field to indicate which page type invokes the conversational servlet. For ConvIterationsResults.jsp the hidden field is indicated as follows:

```
<input type="hidden" name="pageName" value="MiddlePage">
```

Step 4: Modify the Java Servlet, ConvIterations.java

The Java servlet generated by WebSphere Studio's Javabean wizard contains the basic logic to run an IMS transaction, but it must be modified as described below in order to implement the Conversational HttpSession programming model and process the conversation using a single EAB command. The changes are as follows:

- Add additional import statements to the ConvIterations class. See the ConvIterations class in the **com.ibm.connector.ims.sample.conv.phonebook.servlet** package of the Connector IMS Samples project for an example of the additional import statements needed.
- Modify the performTask() method of the ConvIterations class as follows:
  1. Ensure that the servlet uses a single global instance of Connection Manager.

     In general, there should be a single global instance of ConnectionManager per JVM. This example uses the RegisterConnectionManager servlet in package **com.ibm.connector.ims.sample.cm** of the Connector IMS Samples project. The RegisterConnectionManager servlet must be run prior to running this servlet, preferably when the application server is started. The RegisterConnectionManager servlet sets the servlet context with an instance of class com.ibm.connector.connectionmanager.ConnectionManager. This servlet gets that instance from the servlet context and uses it to set the servlet's runtime context.
  2. Obtain the HttpSession object.

     An IMS conversation is made of a series of iterations that spans multiple requests of the servlet. The IMS Connector for Java Conversational HttpSession programming model uses the same HTTP session for all the iterations of a conversation. Since HTTP is a stateless protocol, it does not save any information between one iteration and the next. However, the HttpSession object associated with the HTTP session can be used to save data associated with the conversation between iterations of the conversation. This ensures that the execution context is set up correctly for subsequent requests and that corresponding resources, such as the connection resource, are used appropriately.

     The following shows how to obtain the HttpSession object:

     ```
     // Get the HttpSession object...
     HttpSession httpSession = request.getSession(true);
     ```
  3. Set up for the first iteration of the conversation.
     a. Create a new IMSConvContext object for the new conversation and save it in the HttpSession object for subsequent iterations. In addition, save it in the runtime context. By providing the IMSConvContext object as the session ID for the current runtime context, you ensure that the appropriate connection is used for the current iteration of the conversation.

b. Create a cleanup helper object to be used in case the HTTP session becomes unbound. Save the IMSConvContext object in the cleanup helper object, then save the cleanup helper object in the HttpSession object. Sample code is:

```
IMSConvUnboundHttpSessionCleanup cleanupHelper =
   new IMSConvUnboundHttpSessionCleanup();
cleanupHelper.setConvContext( convContext );
httpSession.putValue("CleanupHelper", cleanupHelper );
```

c. Instantiate the conversational EAB command bean, ConversationalCommand, and store it in the HttpSession object so it can be accessed by called pages.

d. Populate an MI1 object with data from the request object passed to the servlet and use the setInput() method to associate it with the EAB command. Most fields are populated using the getParameter() method of the servlet.

e. Set the EAB command's hostName, port, and datastore properties from the request object and save in the HttpSession object for use in later iterations of the conversation.

f. Set the EAB command's userID, password, and group properties from the request object.

4. Set up for the middle iterations of the conversation.
   a. Get the existing IMSConvContext object for the conversation from the HttpSession object and save it in the runtime context.
   b. Instantiate the conversational EAB command bean, ConversationalCommand, and store it in the HttpSession object so it can be accessed by called pages.
   c. Populate an MI2 object with data from the request object passed to the servlet and use the setInput() method to associate it with the EAB command. Most fields are populated using the getParameter() method of the servlet.
5. Execute the EAB command. This statement is generated by WebSphere Studio.
6. Retrieve the output of the EAB command and determine the type of object returned. Use callPageNamed() to load the appropriate output JSP page to display the data.
   – ConvEndConversation.jsp

     The getConvEnded() method is used to determine if the IMS application program has ended the conversation. After displaying the JSP page with the appropriate information, the HTTP Session is invalidated, and the runtime context is closed and removed from the thread.

   – ConvIterationsResults.jsp

     This JSP page is used to display the output of the iteration of the conversation and, at the same time, allow the user to enter new data to continue the conversation.

   – ConvDFSMessage.jsp

     When a DFS message is returned by IMS in response to a request, it most likely indicates that the request did not complete successfully. For example, a DFS message is returned when the IMS application program abends. The servlet should include the appropriate logic for handling DFS messages, and display a message to the user in the Web browser. At a minimum, the servlet should also invalidate the HTTP session, in order to free up the connection for reuse by another request.

Note that this JSP page invokes the getDFSMessage() method on the ConversationalCommand instance that was saved in the HttpSession object by the ConvIterations servlet. You must save the EAB command bean (the instance of ConversationalCommand) in the HttpSession object if you wish to use it in other than the servlet in which it was instantiated.

7. End the servlet.

   Note that the code for the sample conversational servlet diverges from the template generated by WebSphere Studio. Change your servlet as follows:

   a. *Do not* close the runtime context. The close() method clears the session id (IMSConvContext) associated with the connection and this is needed for the duration of the conversation.

Running the IMS Conversation

You can run the IMS conversation by running your Web application in WebSphere Application Server or in the VisualAge for Java WebSphere Test Environment.

- For instructions on deploying and running a Web application in WebSphere Application Server, see Building a Java Servlet to Run an IMS Transaction.
- For instructions on deploying and running a Web application in VisualAge for Java, refer to *Using the WebSphere Test Environment* under Tasks, and *WebSphere Test Environment* under Concepts in VisualAge for Java's Help pages.

In both cases, for successful execution of the Web application, ensure that the servlet com.ibm.connector.ims.sample.cm.RegisterConnectionManager is run prior to running the Web application.

Chapter 12. Building a Web Application that Uses Two Servlets to Run an IMS Conversation

This section describes how to build a Web application with two servlets that uses IMS Connector for Java to run an IMS conversation. It illustrates an alternate way of implementing the Conversational HttpSession programming model described in Building a Web Application that Uses One Servlet to Run an IMS Conversation. It uses the same IMS application program but the Web application differs in that it uses two servlets instead of one and each servlet executes a separate EAB command. The IMS application program the Web application runs is described in the IMS INSTALL/IVP Sample Application.

Two of the packages in the IMS Connector Samples feature pertain to this example. Java source similar to the Java source for the EAB commands of this example can be found in the **com.ibm.connector.ims.sample.conv.registration** package of the Connector IMS Samples project. Java source similar to the Java source of the servlets for this example is contained in the **com.ibm.connector.ims.sample.conv.registration.servlet** package of the Connector IMS Samples project. These packages are added to your workspace when the IMS Connector Samples feature is added. See Preparing your VisualAge for Java Environment for instructions on installing the IMS Connector Samples feature if you have not already done so.

In addition, other Web application files such as the HTML and JSP files, JAR files, the SERVLET and CLASS files in their appropriate folder(s), and a theme folder containing Master.css can be found in <*IBM_Connectors_install_dir*>\imsconn\samples\servlet\registration, where <*IBM_Connectors_install_dir*> is the IBM Connectors installation directory on the drive on which VisualAge for Java is installed.

By following the instructions in this section, you will create an application which is the same as the corresponding sample, except that it uses different project and package names than are used in the sample since you can't have duplicate package or project names within your VisualAge environment. As a result, you will be able to check your work against the sample, helping you to resolve any difficulties you encounter as you progress through the steps that follow.

Creating Input and Output Record Beans for an IMS Conversation

Java record beans for an IMS conversation are created by VisualAge for Java from the COBOL source file(s) that contain the data structures (01 commareas) for the input and output messages of each iteration of the IMS conversation. These file(s) are provided to VisualAge for Java's COBOL importer.

For this example, data structures for all the iterations of the conversation are found in the single IMS application program, dfsiva34.mem, which can be found in <*IBM_Connectors_install_dir*>\imsconn\samples\misc, where <*IBM_Connectors_install_dir*> is the IBM Connectors installation directory.

For the IMS conversation of this sample, the iterations of the conversation are the same as described in Creating Input and Output Record Beans for an IMS Conversation under Building a Web Application that Uses One Servlet to Run an IMS Conversation.

© Copyright IBM Corp. 1998, 2000

The record beans representing the input and output messages used by this Web application and steps to build them are also the same as described in Creating Input and Output Record Beans for an IMS Conversation (page 107) under Building a Web Application that Uses One Servlet to Run an IMS Conversation. However, you may wish to create a different package for the record beans and EAB commands of this Web application; for example, convcmds2.

Building EAB Commands for the IMS Conversation

This section describes how you create the two Enterprise Access Builder (EAB) commands for this example's IMS conversation. For a description of the different ways in which you could model your Web application, see Building an EAB Command for the IMS Conversation (page 113) under Building a Web Application that Uses One Servlet to Run an IMS Conversation.

The Web application for this example is modeled using two EAB commands:

- The first EAB command, ConversationalCommand1, models the first iteration of the IMS conversation. It consists of a record bean for the initial input message, MI1, a record bean for the output message, MO2, an IMSConnectionSpec object, and an IMSInteractionSpec object. This EAB command is executed by the first servlet, ConvIterations1.
- The second EAB command, ConversationalCommand2, models subsequent (middle) iterations of the IMS conversation. It consists of a record bean for the input message, MI2, a record bean for the output message, MO2, an IMSConnectionSpec object, and an IMSInteractionSpec object. This EAB command is executed by the second servlet, ConvIterations2.

The steps for creating the EAB commands for this example are presented below. The EAB command is created in the same project and package as the input and output record beans described above.

1. Create an EAB Command Class for the IMS Conversation
2. Promote the Connection Specification Properties
3. Promote the Interaction Specification Properties
4. Promote the Input Record Bean Properties
5. Promote the Output Record Bean Properties
6. Create the Command Bean

Step 1: Create an EAB Command Class for the IMS Conversation

For a description of how to create a command class, see Step 1: Create a Command Class for the IMS Transaction (page 40) under Creating an EAB Command Using the Command Editor (page 39) in Building a Java Application to Run an IMS Transaction.

For this particular example, the first EAB command should be created as follows:

- Use ConversationalCommand1 for the name of the EAB command class
- Add com.ibm.connector.imstoc.IMSConnectionSpec to the EAB command class
- Add com.ibm.connector.imstoc.IMSInteractionSpec to the EAB command class
- Add input bean convcmds2.MI1 to the EAB command class
- Add output bean convcmds2.MO2 to the EAB command class
- Add output bean com.ibm.connector.imstoc.DFSMsg to the EAB command class

For this particular example, the second EAB command should be created as follows:

- Use ConversationalCommand2 for the name of the EAB command class
- Add com.ibm.connector.imstoc.IMSConnectionSpec to the EAB command class
- Add com.ibm.connector.imstoc.IMSInteractionSpec to the EAB command class
- Add input bean convcmds2.MI2 to the EAB command class
- Add output bean convcmds2.MO2 to the EAB command class
- Add output bean com.ibm.connector.imstoc.DFSMsg to the EAB command class

Step 2: Promote the Connection Specification Properties

For a description of how to promote connection specification properties, see Step 6: Promote the Connection Specification Properties (page 45) under Creating an EAB Command Using the Command Editor (page 39) in Building a Java Application to Run an IMS Transaction. For this particular example, promote the following properties for IMSConnectionSpec in each EAB command:

- Port
- Host name

Step 3: Promote the Interaction Specification Properties

For a description of how to promote interaction specification properties, see Step 5: Promote the Interaction Specification Properties (page 44) under Creating an EAB Command Using the Command Editor (page 39) in Building a Java Application to Run an IMS Transaction. For this particular example, promote the following properties for IMSInteractionSpec in each EAB command:

- Datastore name
- Conversation ended

Step 4: Promote the Input Record Bean Properties

For a description of how to promote input message bean properties, see Step 2: Promote the IMS Transaction Input Properties (page 42) under Creating an EAB Command Using the Command Editor (page 39) in Building a Java Application to Run an IMS Transaction.

For this particular example, for MI1 of ConversationalCommand1, promote the following properties:

- IN__LL
- IN__FIRST__NAME
- IN__EXTENSION
- IN__LAST__NAME
- IN__ZIP__CODE
- IN__COMMAND
- IN__TRANCD
- IN__ZZ

For this particular example, for MI2 of ConversationalCommand2, promote the following properties:

- IN__LL
- IN__FIRST__NAME
- IN__EXTENSION

- IN__LAST__NAME
- IN__ZIP__CODE
- IN__COMMAND
- IN__ZZ

Step 5: Promote the Output Record Bean Properties

For a description of how to promote output message bean properties, see Step 3: Promote the IMS Transaction Output Properties (page 43) under Creating an EAB Command Using the Command Editor (page 39) in Building a Java Application to Run an IMS Transaction.

For this particular example, for MO2 of ConversationalCommand1 and ConversationalCommand2, promote the following properties:

- OUT__FIRST__NAME
- OUT__MESSAGE
- OUT__LAST__NAME
- OUT__EXTENSION
- OUT__ZIP__CODE

Step 6: Create the Command Bean

For a description of how to create the command bean, see Step 7: Create the Command Bean (page 45) under Creating an EAB Command Using the Command Editor (page 39) in Building a Java Application to Run an IMS Transaction.

Note: For this example, the getDFSMessage() method of com.ibm.connector.imstoc.DFSMsg is added to the EAB command classes (ConversationalCommand1 and ConversationalCommand2). For a description of adding this method see Step 4: Promote the DFS Message Properties (page 43) under Creating an EAB Command Using the Command Editor (page 39) in Building a Java Application to Run an IMS Transaction.

Creating a Conversational Web application

After the EAB command beans are built, they are exported in the form of a JAR file. The JAR file is used by WebSphere Studio to generate a template for the Web application, and at runtime for access to the classes of the EAB command.

Use VisualAge for Java to create the JAR file. For a description of how to create a JAR file for an EAB command, see Step 2: Provide Your EAB Command to WebSphere Studio (page 95) in Creating a Java Servlet Using WebSphere Studio (page 95) under Building a Java Servlet to Run an IMS Transaction. When creating the JAR file for this example, mark the classes ConversationalCommand1 and ConversationalCommand2 as a beans.

The following sections summarize the steps involved in developing the conversational Web application. The reader may wish to examine the servlets in package **com.ibm.connector.ims.sample.conv.registration.servlet** of the Connector IMS Samples project while developing their own Web application.

Step 1: Use WebSphere Studio to Create a Web Application Template

WebSphere Studio does not create an exact template for this two-servlet example. However, you can create a Web application template that approximates the one needed for the final Web application by using the following process:

- In WebSphere Studio, create a project for your Web application, for example convphonereg.
- Insert the JAR file described above in the servlet folder of your project and start the **JavaBean Wizard**.

The next step involves making two passes through WebSphere Studio's **JavaBean Wizard**. For the first pass:

- On the first page of the **JavaBean Wizard**, select ConversationalCommand1.
- Choose to create an input page, a results page, and an error page.
- On the Input Page of the **JavaBean Wizard** select fields *hostName*, *portNumber*, *dataStoreName*, *IN__FIRST__NAME*, *IN__EXTENSION*, *IN__LAST__NAME*, *IN__ZIP__CODE*, *IN__COMMAND*. Additional fields and more appropriate captions are added to the output JSP page later.
- On the Results Page of the **JavaBean Wizard** select fields *OUT__EXTENSION*, *OUT__FIRST__NAME*, *OUT__LAST__NAME*, *OUT__MESSAGE*, and *OUT__ZIP*. Additional fields and more appropriate captions are added to the output JSP page later.
- On the Methods page of the **JavaBean Wizard** select the *execute()* method.
- On the Session page of the **JavaBean Wizard**, in answer to *Will you use this bean or query on more than one page?*, select the *Yes, store it in the user's session()* radio button.
- On the Finish page of the **JavaBean Wizard** select the *Rename...* button and choose ConvIterations1 as the prefix for the generated file names.

  For the second pass:

  – On the first page of the **JavaBean Wizard**, select ConversationalCommand2.
  – Choose to create an input page, a results page, and an error page.
  – On the Input Page of the **JavaBean Wizard** select fields *hostName*, *portNumber*, *dataStoreName*, *IN__FIRST__NAME*, *IN__EXTENSION*, *IN__LAST__NAME*, *IN__ZIP__CODE*, *IN__COMMAND*. Additional fields and more appropriate captions are added to the output JSP page later.
  – On the Results Page of the **JavaBean Wizard** select fields *OUT__EXTENSION*, *OUT__FIRST__NAME*, *OUT__LAST__NAME*, *OUT__MESSAGE*, and *OUT__ZIP*. Additional fields and more appropriate captions are added to the output JSP page later.
  – On the Methods page of the **JavaBean Wizard** select the *execute()* method.
  – On the Session page of the **JavaBean Wizard**, in answer to *Will you use this bean or query on more than one page?*, select the *Yes, store it in the user's session()* radio button.
  – On the Finish page of the **JavaBean Wizard** select the *Rename...* button and choose ConvIterations2 as the prefix for the generated file names.

    The following files are generated by WebSphere Studio and are used as a template for the Web application:

    - ConvIterations1Input.html
    - ConvIterations1.java
    - ConvIterations1Results.jsp
    - ConvIterations1Error.jsp
    - ConvIterations1.servlet

- ConvIterations2Input.html
- ConvIterations2.java
- ConvIterations2Results.jsp
- ConvIterations2Error.jsp
- ConvIterations2.servlet
- Master.css, in a folder called *Theme*

Step 2: Modify the Web Application Template

The Web application template must be modified extensively in order to create a conversational Web application. Files must be modified and new JSP pages must be added. The following summarizes the modified and new files of the two-servlet conversational Web application:

- ConvIterations1Input.html

  This page is modified to allow the user to provide the data for the input message of the first iteration of the conversation, MI1. It invokes the first servlet, ConvIterations1.

  Fields hostName, portNumber, dataStoreName, userID, password, and group are also included and new captions and positioning are provided. These 6 fields are only be provided for input on the initial HTML page.

- ConvIterations1Results.jsp

  This page is modified to display the output message of the first iteration of the conversation. In addition, it is modified to allow the user to provide the data for the input message of the next iteration of the conversation, MI2, and to invoke the second servlet, ConvIterations2.

  The basic structure of this page is changed from an output JSP page to a combination of an output JSP page, to display the data of the output message of the first iteration (MO2), and a data input form, to accept the data for the input message of the next iteration (MI2). See Step 3: Modify the Results JSP Page, ConvIterationsResults.jsp (page 117) in Creating a Conversational Web application (page 115) under Building a Web Application that Uses One Servlet to Run an IMS Conversation for a description of changing an output JSP page to a data input form.

- ConvIterations1.java

  This servlet is modified, to implement the Conversational HttpSession programming model, to be invoked by a request object containing data for the first iteration of the conversation, and to display a page with the results of the first iteration (ConvIterationsResults1.jsp), a DFS message page (ConvCmd1DFSMessage.jsp), or a page for the end of the conversation, (ConvEndConversation.jsp). See Step 4: Modify the Java Servlet, ConvIterations.java (page 118) in Creating a Conversational Web application (page 115) under Building a Web Application that Uses One Servlet to Run an IMS Conversation for a description of the type of changes that need to be made to the servlet. The ConvIterations1 servlet only processes the first iteration of the conversation.

- ConvIterations2.java

  This servlet is modified, to implement the Conversational HttpSession programming model, to be invoked by a request object containing data for middle iterations of the conversation, and to display a page with the results of middle iterations (ConvIterationsResults2.jsp), a DFS message page (ConvCmd2DFSMessage.jsp), or a page for the end of the conversation, (ConvEndConversation.jsp). See Step 4: Modify the Java Servlet, ConvIterations.java (page 118) in Creating a Conversational Web application

(page 115) under Building a Web Application that Uses One Servlet to Run an IMS Conversation for a description of the type of changes that need to be made to the servlet. The ConvIterations1 servlet only processes middle iterations of the conversation.

- ConvIterations2Results.jsp

  This page will be modified to display the output message a middle iteration of the conversation. In addition, it will be modified to allow the user to provide the data for the input message of the subsequent iterations of the conversation, MI2, and to invoke the second servlet, ConvIterations2.

  The basic structure of this page is changed from an output JSP page to a combination of an output JSP page, to display the data of the output message of a middle iteration (MO2), and a data input form, to accept the data for the input message of the next iteration (MI2). See Step 3: Modify the Results JSP Page, ConvIterationsResults.jsp in Creating a Conversational Web application (page 115) under Building a Web Application that Uses One Servlet to Run an IMS Conversation for a description of changing an output JSP page to a data input form.

- ConvCmd1DFSMessage.jsp

  This page is added to the Web application. It is used by the first servlet, ConvIterations1, to display any ″DFS″ messages returned by IMS while the IMS conversation is running.

- ConvCmd2DFSMessage.jsp

  This page is added to the Web application. It is used by the second servlet, ConvIterations2, to display any ″DFS″ messages returned by IMS while the IMS conversation is running.

- ConvEndConversation.jsp

  This page is added to the Web application. It displays a separate page indicating that the IMS conversation has ended.

- ConvIterationsError.jsp

  This is the error page generated by WebSphere Studio. No changes are made to this file.

- ConvIterations1.servlet

  This file, the servlet configuration file for the first servlet, is modified to allow the callPageNamed() method to be used to display the JSP pages described above.

- ConvIterations2.servlet

  This file, the servlet configuration file for the second servlet, is modified to allow the callPageNamed() method to be used to display the JSP pages described above.

- Master.css, in a folder called *Theme*

  This is the style sheet generated by WebSphere Studio. No changes are made to this file.

Running the IMS Conversation

You can run the IMS conversation by running your Web application in WebSphere Application Server or in VisualAge for Java.

- For instructions on deploying and running a Web application in WebSphere Application Server, see Building a Java Servlet to Run an IMS Transaction.

- For instructions on deploying and running a Web application in VisualAge for Java, refer to *Using the WebSphere Test Environment* under Tasks, and *WebSphere Test Environment* under Concepts in VisualAge for Java's Help pages.

In both cases, for successful execution of the Web application, ensure that the servlet com.ibm.connector.ims.sample.cm.RegisterConnectionManager is run prior to running the Web application.

Appendix A. IMS INSTALL/IVP Sample Application

The IMS transactions described in this section, as well as in other sections, are transactions that are part of the IMS INSTALL/IVP Sample Application. Two IMS INSTALL/IVP Sample Application transactions are used in the IMS Connector Samples and documentation. They are IVTNO which is a nonconversational transaction and IVTCB which is a conversational transaction. If these transactions are not installed on your IMS system, contact your IMS system administrator to have them installed. The INSTALL/IVP Sample Application is packaged and shipped with IMS. These transactions, as they are used in this documentation, have the following characteristics:

- The IVTNO transaction is nonconversational and the IVTCB transaction is conversational.
- The transaction codes are IVTNO (nonconversational) and IVTCB (conversational.).
- The IMS INSTALL/IVP Sample Application transactions, both conversational and nonconversational, accept a single-segment input message and return a single-segment output message.
- The IMS INSTALL/IVP Sample Application transactions support ADD, DISPLAY, and DELETE commands for working with entries in a phone book database.

Typically, you first add an entry to the database using the **ADD** command. You must provide values for all of the fields of the single-segment IMS transaction input message, including LL and ZZ. You can do this by invoking the appropriate "set" methods of the EAB command. For example:

```
cmd.setInMsg1IN__LL( (short) ((InMsg)cmd.getInput()).getSize() );
cmd.setInMsg1IN__ZZ( (short)0 );
cmd.setInMsg1IN__TRCD( "IVTNO" );
cmd.setInMsg1IN__CMD(  "ADD" );
cmd.setInMsg1IN__NAME1( "Bond" );
cmd.setInMsg1IN__NAME2( "James" );
cmd.setInMsg1IN__EXTN( "1234" );
cmd.setInMsg1IN__ZIP( "99000" );
```

You must always provide the correct **LL** value. Each IMS message segment (either input or output) begins with a 2-byte segment length (**LL**) field, followed by a 2-byte flag (**ZZ**) field. The **ZZ** field can be set to zero on input, but the input value for **LL** must be the sum of the defined lengths of the input fields plus 4 (corresponding to the length of the entire message, i.e., the sum of the lengths of the input fields plus the lengths of the **LL** and **ZZ** fields.)

After an entry is added to the database, you can display it by using the **DISPLAY** command, as follows:

```
cmd.setInMsg1IN__LL( (short) ((InMsg)cmd.getInput()).getSize() );
cmd.setInMsg1IN__ZZ( (short)0 );
cmd.setInMsg1IN__TRCD( "IVTNO" );
cmd.setInMsg1IN__CMD(  "DISPLAY" );
cmd.setInMsg1IN__NAME1( "Bond" );
cmd.setInMsg1IN__NAME2( "" );
cmd.setInMsg1IN__EXTN( "" );
cmd.setInMsg1IN__ZIP( "" );
```

The names of the actual "set" methods in the examples will vary, depending upon the names of the corresponding transaction input and output message beans.

© Copyright IBM Corp. 1998, 2000

The examples in other sections of the documentation illustrate this database in different states. For instance, some sections assume that the entry being displayed has been previously added to the database.

The COBOL source for the input and output messages of the nonconversational transaction can be found in \imstoc\samples\misc\Ex01.ccp, where is the IBM Connectors installation directory. For example, d:\IBM Connectors. Initially, <*IBM_Connectors_install_dir*> is <vaj_install_drive>\IBM Connectors, where <vaj_install_drive> is the drive on which VisualAge for Java is installed. The COBOL source for the input and output messages of the conversational transaction can be found in \imstoc\samples\misc\dfsiva34.mem.

Appendix B. IMS Connector for Java Conversational Programming Models

IMS Connector for Java Conversational Support provides customers with two programming models from which to build Java applications and Web applications that iterate through the interactions of an IMS conversational program. The programming model is based on the existing application model that uses IMS Connector for Java to invoke IMS transactions via IMS Connect. The application is built using VisualAge for Java and WebSphere Studio. VisualAge for Java tools have been enhanced to help customers generate the record components specifically needed for generating the input and output messages of an IMS conversational transaction.

Generally, a Java application is built with a GUI interface that consists of multiple window panels. The user provides the input data and views the output data within separate panels of the same Java application when stepping through the iterations of the IMS conversation. The Java application contains logic that utilizes Java classes in the IMS Connector for Java Conversational Support to invoke the IMS conversational transactions.

A Web application is made up of a group of application components that can be accessed from the Web and consists of one or more Java servlets that contain the application logic, an HTML page which serves as the initial input page used to start the application and one or more JSPs (Java Server Pages) for obtaining input data from and displaying output data to the user. The Java servlets within a Web application share the same servlet context. The user starts an IMS conversational transaction from a web browser by first providing some input data via an HTML form. The web server then invokes the Java servlet running inside the WebSphere Application Server. The servlet utilizes the Java classes in IMS Connector for Java (including the Conversational Support classes) to send the conversational transaction request and receive the transaction output via IMS Connect. The output will be displayed back to the user through a JSP in the web browser.

Although this documentation focuses primarily on modeling an IMS conversation with a Web application, many of the same rules that pertain to Web applications can be applied to building Java applications, as well. Generally, the rules governing the logic of a Java servlet can be applied to the logic of a Java application. Similarly, the rules governing the use of html and JSP pages can be applied to the GUI interfaces of a Java application.

Conversational HttpSession Model

The Conversational HttpSession Model for building a Web application to invoke an IMS conversational transaction is similar to the programming model used for building a Web application to invoke a non-conversational IMS transaction. The steps involve the use of VisualAge for Java and WebSphere Studio and are as follows:

1. Provide your COBOL source files to VisualAge for Java to generate the input and output Java record beans
2. Use the VisualAge for Java Command Editor to create an Enterprise Access Builder (EAB) command

© Copyright IBM Corp. 1998, 2000

3. Input the EAB command to WebSphere Studio and use WebSphere Studio to generate a template for a Web application. Build a Web application from the template using the guidelines outlined below. This process can be summarized as follows:
   a. Create additional JSP pages for the iterations
   b. Modify the JSP pages
   c. Modify the Java servlet

IMS Connector for Java Conversational Support introduces a session-enabled Web application model for an IMS conversational transaction, the Conversational HttpSession Model. A session encompasses a series of requests originating from the same browser to the Java servlet or servlets of a Web application. A session is analogous to an SNA connection between a terminal user and an IMS conversational application program.

The user submits a series of request from the same browser to iterate through different interactions of an IMS Conversation. The user starts an IMS conversational transaction from a web browser by submitting a request to IMS to schedule (or run) a conversational transaction. The HTTP session associated with the browser is used to group the series of interactions from that browser. The conversation continues through any number of iterations until it is terminated, either by a user action (i.e., the client) or by transaction program logic that directs IMS to end the conversation at the conclusion of that iteration. The conversational transaction can also be terminated automatically by IMS Connector for JavaÆs classes whenever the session-enabled Web application times out, for example, when the browser remains idle for a specified period of time. This timeout is set using the JavaServlet APIs (the default is 30 minutes.)

The JavaServlet specification provides a class, HttpSession, that is used by WebSphere Application Server to create an object associated with the HTTP session used by the Web application. Once a session is established between the browser and the Java servlet, the Java servlet maintains a reference to this unique HttpSession object (i.e., the HttpSession object is bound to the servlet) throughout all of the iterations for the duration of the conversation. The Java servlet uses this unique HttpSession object to obtain information about the state of the session as well as the state of the conversation that is running in that session.

An HttpSession object is associated with (bound to) a servlet at the start of the first iteration of an IMS conversation. It remains bound to that servlet or a succeeding servlet in the case of a multi-servlet Web application until the servlet detects that either the user has taken some action which will cause the IMS conversation to be terminated or the IMS application program has terminated the conversation. The session should be unbound if there are any errors encountered in the application flow. Possible errors include communication failure between components and abnormal termination of components. The Java servlet handles these errors appropriately. Generally, when an error has occurred, the conversation should be terminated and the connection resource cleaned up for reuse.

For example, a browser session becomes unbound from a servlet when the browser times out waiting for input from the user. When this happens, the servlet should terminate the conversation and clean up the CCF connection resources. IMS Connector for Java Conversational Support provides the IMSConvHttpSessionCleanup class to capture the notification from the HttpSession object when the HttpSession object becomes unbound. The IMSConvHttpSessionCleanup object instantiated by the servlet will then terminate the conversational transaction and clean up the connection resources appropriately.

Since the HTTP protocol, which is used as the connection mechanism between the web browser and the Java
servlet, is stateless, it is often useful to bind objects to the HttpSession object in order to save intermediate state data of the Web application. For example, you may want to save information about the current iteration of the IMS conversation for use when the next iteration of the conversation is executed.

Another use for the HttpSession object is to save information about the connection being used for a particular conversation. This is necessary since IMS Connect requires a dedicated connection for the entire duration of an IMS conversation. In other words, IMS Connect requires that the same connection be used for all of the communications between the Web application (Java servlet or servlets) and IMS Connect. Furthermore, that connection cannot be used for any other communications until that conversation has ended. A connection for purposes of this discussion, is the communications link between the Java application or servlet and IMS Connect. In a TCP/IP communications environment, a socket is this connection (or, to be more precise, represents this connection).

In order to provide a mechanism for saving information about the connection being used for an IMS conversation in the HttpSession object, IMS Connector for Java includes the IMSConvContext class in its Conversational HttpSession programming model for conversational Web applications. This class provides a token which is stored in the HttpSession object and used to bind a particular CCF connection to a servlet that is part of an IMS Conversation. Therefore this servlet needs to create a single IMSConvContext object at the start of each IMS conversation. This object is then stored in the HttpSession object. The HttpSession object and therefore the IMSConvContext are then used by the CCF ConnectionManager to initially bind a conversation to a particular connection and later to ensure that the CCF ConnectionManager will use that same connection for each iteration of the conversation. When the HttpSession object becomes unbound or an error occurs during the conversation, or at the end of the conversation, the IMSConvContext reference must be unbound from its connection in order to allow the ConnectionManager to reuse that connection for new requests.

When the HttpSession object becomes unbound this is handled automatically by the IMSConvHttpSessionCleanup object that was instantiated in the Web application servlet when the conversation began. In the case of an error being detected in a servlet during a conversation, the unbinding is handled through the use of the httpSession.invalidate() which in turn causes the IMSConvHttpSessionCleanup object to be invoked. When a conversation ends normally, the IMSConvContext reference is unbound from its connection through the use of the RuntimeContext.getCurrent().close() method.

When the conversational transaction is completed, a new conversational or non-conversational transaction can be run next from the client. However, if the IMS conversation is not completed, then you cannot run a new transaction with the same connection resource.

In the Conversational HttpSession model, the application flow for an IMS conversational transaction starts when the user initiates a conversation from the browser. The flow continues with a series of interactions between the Web application and the IMS conversational application program performing the business logic of the application. Finally, the conversation is terminated by the user or by the IMS application program. The application flow is illustrated as follows:

Iterating in the IMS Conversation

1. The user starts an IMS conversational Web application by displaying the input HTML page of the Web application in a Web browser.

2. The user provides the initial input data of the IMS conversational transaction in the input HTML page. This starts the first iteration of the IMS conversational transaction.

3. The Web browser sends the request with the input data to a Java servlet running in WebSphere Application Server.

4. The Java servlet obtains the HttpSession object and creates an IMSConvContext object which it saves in that HttpSession object for use by the CCF ConnectionManager to ensure that the same connection is used throughout a conversation. The servlet then invokes the execute() method of the EAB command, passing the input data received from the browser. The EAB command uses IMS Connector for Java APIs to send the transaction request to IMS via IMS Connect.

5. IMS receives the request and schedules the IMS conversational application program. The IMS application program processes the request and the reply is returned to the Java servlet.

6. The Java servlet populates the EAB command with the output data of the reply and loads the appropriate result JSP page to display the output data on the Web browser. The Java servlet or the end user determines if the output data received is acceptable. If it is, the Java servlet invokes the commit() method of JavaCoordinator to accept the output and commit that iteration of the conversation (by sending an ACK message to IMS Connect). Otherwise, it invokes the rollback() method of JavaCoordinator to reject the output and back out that iteration of the conversation (by sending a NACK message to IMS Connect.)

7. The user provides the data for the second iteration on the JSP page used to display the output data of the first iteration, then submits the next request to IMS. This invokes the second iteration of the IMS conversational transaction.

Subsequent iterations of the conversation repeat steps 3-7 until the user or the IMS application program terminates the conversation. In step 4, on the second and all subsequent iterations of the conversation, the Java servlet obtains the IMSConvContext object from the its HttpSession object.

Note: The steps described above may differ when used with different synchronization levels. The above scenario describes an application flow with SYNC_LEVEL_CONFIRM. When used with SYNC_LEVEL_NONE, the call to the JavaCoordinator commit() or rollback() method in step 6 will not be made.

Terminating an IMS Conversation

In the IMS Connector for Java Conversational HttpSession model, an IMS conversation is typically terminated in one of two ways:

1. The client terminates the IMS conversational transaction:
    a. The end user submits a request to end the IMS conversation from the browser.
    b. The Java servlet invokes an EAB command with the MODE_END_CONVERSATION interaction mode. MODE_END_CONVERSATION is a "send only" interaction. IMS

terminates the conversation. The transaction is not scheduled and no output message is sent to the servlet from IMS.

c. The Java servlet should invalidate the HttpSession object and clean up the connection resource for reuse.

d. The servlet should also load a customized JSP page to display a message indicating an attempt was made to end the conversation.

2. The IMS application program terminates the conversation:
   1. The end user submits a request to IMS to run the IMS conversational transaction. The input message may contain a specific request to terminate the conversation. (The IMS application program may also terminate the conversation based on criteria other than the contents of the input message.)
   2. IMS schedules the IMS conversational application program, the program processes the input message, then sends the transaction output message back to the servlet. The OTMA message containing the transaction output message contains a bit signifying that the conversation has been ended. IMS ends the conversation after sending the transaction output message back to the servlet.
   3. IMS Connector for Java, receives the OTMA message and sets the value of the *convEnded* property in the IMSInteractionSpec to true. The EAB command is populated with the output data.
   4. The Java servlet should invalidate the HttpSession object which, among other things, will clean up the connection resource.
   5. The servlet should also load a customized JSP page to display a message on the Web browser indicating that the conversation was ended.

Conversational Navigator Model

The Conversational Navigator Programming Model for building a Web application to invoke an IMS conversational transaction is similar to the programming model for building a Web application to invoke a non-conversational IMS transaction. The steps involve the use of VisualAge for Java and WebSphere Studio and are as follows:

1. Provide your COBOL source file(s) to VisualAge for Java to generate the input and output Java record beans.
2. Use the VisualAge for Java Command Editor to create an Enterprise Access Builder (EAB) command for each iteration of the IMS conversation.
3. Using the VisualAge for Java Visual Composition Editor, create an EAB Navigator.
4. Generate a Web application using WebSphere Studio and VisualAge for Java.
5. Customize the HTML/JSP pages.
6. Modify the Java servlet

An EAB Navigator can be used to model any IMS conversation where all of the data required for all of the interactions that make up that particular conversation is known in advance. A conversational Navigator consists of EAB Commands and/or other Navigators linked together to form a more complex Navigator which encapsulates all of the interactions that make up a given IMS conversation. Each component EAB Command and Navigator within the conversational Navigator is then executed in the pre-determined order programmed into the conversational Navigator. Each iteration of an IMS conversation could be modeled as an EAB command. Then, a navigator is built to encapsulate a sequence of EAB command iterations to model the whole IMS conversation for a particular task.

When you execute a conversational EAB Navigator, as with any other EAB Navigator, it provides its input to the EAB Commands and Navigators of which it is comprised. The output of the component EAB Commands and Navigators is available to the conversational Navigator to be used as output, again depending on how the conversational Navigator is programmed.

Once a conversational Navigator has started executing, control will not be returned to the user until all of the component EAB Commands and Navigators (which represent the iterations of the conversation) have completed and the conversation has ended. No user interaction with an EAB Navigator is possible once execution of the Navigator has started. There is no means available to provide additional data to be used by subsequent Navigator components after a given Navigator component has finished execution. It is for this reason that all of the input data needed by each component EAB Command or Navigator must be provided to the conversational Navigator before it begins executing.

Considerations in choosing a model

You should choose the Navigator model if it is suitable for your IMS conversation since it provides a simpler development paradigm and a more effective use of connections. From a development perspective, the conversational Navigator model eliminates the need to deal with issues such as using the ″back″ button, bookmarking, browser caching or tracking the state of the IMS conversation. Issues such as scaling, cloning, and clustering may affect your decision of which model to use. Consult the WebSphere Application Server documentation for a detailed discussion of these issues.

The Conversational Navigator model is suitable when a particular sequence of interactions representing a well defined task and the data required by all of those interactions can be pre-determined. A conversational IMS application can be used to process any number of tasks some of which may be suitable for the Navigator model and some of which are not. You will probably find that some small number of tasks for a given IMS conversational application can be implemented with conversational Navigators but the majority will require the HttpSession based servlets.

Appendix C. Using the trace and error logging facility

The trace and error logging facility is implemented by the RASService interface class. It consists of two different output streams. All trace information is logged to the trace output stream, and the errors and exceptions are logged to the error output stream. The levels of trace information that are logged are as follows:

RAS_TRACE_OFF
No trace information is logged. This is the default trace level.

RAS_TRACE_ERROR_EXCEPTION
Only errors and exceptions are logged.

RAS_TRACE_ENTRY_EXIT
Entry and exit of methods, errors, and exceptions are logged.

RAS_TRACE_INTERNAL
The internal state of a connector's object, entry and exit of methods, errors, and exceptions are logged.

RAS_TRACE_INTERNAL_NATIVE
Displays most of the information, including the internal state of other objects used by the connector and all the information in the above trace levels.

You can set the trace level in your code when using the JavaRASService implementation class. Replacing the DefaultRASService with the JavaRASService allows for better control of the error and trace reporting mechanisms. To change to use JavaRASService, you need to execute the following code:

```
com.ibm.connector.infrastructure.java.JavaRASService ras =
    new com.ibm.connector.infrastructure.java.JavaRASService();
runtimeContext.setRASService(ras);
```

Note: JavaRASService is the default implementation class of the RASService for the JavaRuntimeContext class. Therefore, you do not need to implement the above code if you are using JavaRuntimeContext in your application.

The following sample code sets the trace level of the JavaRASService trace facility to log the entry and exit of each method:

```
((com.ibm.connector.infrastructure.java.JavaRASService)runtimeContext.getRASService()).setTraceLev
 com.ibm.connector.infrastructure.RASService.RAS_TRACE_ENTRY_EXIT );
```

Errors are always logged to the error log stream.

Related Reading: For information on the methods of the com.ibm.connector.infrastructure.RASService interface, see its Javadoc information.

Customizing the trace facility

By default, the RASService sends the trace log to the System.out stream, and sends the error log to the System.err stream. You can use the com.ibm.connector.infrastructure.java.JavaRASService class to direct the error or trace log to a different output stream (such as a file). The JavaRASService class provides the following methods to customize the output stream of the error and trace that the connector uses:

© Copyright IBM Corp. 1998, 2000

```
public void setErrorStream(java.io.OutputStream)
Set the output stream of the error log
public void setTraceStream(java.io.OutputStream)
Set the output stream of the trace log
```

Related Reading:
For information on the methods of com.ibm.connector.infrastructure.java.JavaRASService class, see the Javadoc information for the class.

For information on using the trace facility in the WebSphere Application Server environment, see the WebSphere Application Server documentation.

Reading the trace log

Depending on the trace level that is set, the trace log can contain different information:

RAS_TRACE_ERROR_EXCEPTION

The information at this level is similar to that in the error log. See Reading the Error Log for more information.

RAS_TRACE_ENTRY_EXIT

At this level, the trace log shows the entry and exit of a method, such as the following:

—> [com.ibm.connector.imstoc.connectionSpec@2g3.getHostName()]
The right-arrow symbol (—>) represents the entry of a method. The left-arrow symbol (<—) represents the exit of a method. All the information within the section of a —> and a <— of the same method shows all of the execution call stack within that method.

A double arrow symbol (<—>) represents that the method has been entered and exited. No specific information within the method is logged. The name before the @ sign shows the class name. The alphanumeric number after the @ sign shows the hash code of the class. The name after the '.' of the hash code shows the method name.

RAS_TRACE_INTERNAL

At this level, the internal state of an object and other important information are dumped to the trace log. The dump of an object normally starts with the name of the class to be dumped in brackets []. This is similar to the entry/exit, as shown above, but without any symbol at the beginning. It looks similar to the following:

```
[com.ibm.connector.imstoc.communication@1c27]
```

Each field value of the class is shown.

RAS_TRACE_INTERNAL_NATIVE

At this level, more information about the internal state of other objects being used by the connector are dumped to
the trace log. The dump format of the object is similar to the RAS_TRACE_INTERNAL trace level.

Reading the error log

When an exception occurs, the exception information and a printstack of the exception is sent to the error log stream. The following is an example of the error log:

```
com.ibm.connector.CommunicationException:
HWSJ003E: com.ibm.connector.imstoc.IMSAdapter@77cb.connect() error.
Fails to connect to Host name [CSDMEC15],
Port [9999]. [java.net.SocketException: Connection timed out]
  java.net.SocketException: Connection timed out
  java.lang.Throwable(java.lang.String)
  java.lang.Exception(java.lang.String)
  java.io.IOException(java.lang.String)
  java.net.SocketException(java.lang.String)
  void java.net.PlainSocketImpl.socketConnect(java.net.InetAddress, int)
  void java.net.PlainSocketImpl.doConnect(java.net.InetAddress, int)
  void java.net.PlainSocketImpl.connectToAddress(java.net.InetAddress, int)
  void java.net.PlainSocketImpl.connect(java.net.InetAddress, int)
  java.net.Socket(java.net.InetAddress, int, java.net.InetAddress, int, boolean)
  java.net.Socket(java.lang.String, int)
  void com.ibm.connector.imstoc.IMSAdapter.connect()
  void com.ibm.connector.imstoc.IMSConnection.connect()
  void com.ibm.connector.imstoc.IMSCommunication.connect()
  void com.ibm.ivj.eab.command.CommandCommunicationPrimitive.
    beforeExecute(com.ibm.ivj.eab.command.CommandEvent)
  void com.ibm.ivj.eab.command.CommunicationCommand.
    connEtoM1(com.ibm.ivj.eab.command.CommandEvent)
  void com.ibm.ivj.eab.command.CommunicationCommand.
    beforeInternalExecution(com.ibm.ivj.eab.command.CommandEvent)
  void com.ibm.ivj.eab.command.Command.
    fireBeforeInternalExecution(com.ibm.ivj.eab.command.CommandEvent)
  void com.ibm.ivj.eab.command.Command.
    execute(com.ibm.ivj.eab.command.CommandEvent)
  void com.ibm.ivj.eab.command.Command.execute()
  void com.ibm.ivj.eab.command.CommunicationCommand.execute()
  void JTCV36.JTCV36Execute.main(java.lang.String [])
```

This information consists of the error message of the exception and the printstack information of the exception. First, it shows the exception that is being thrown (com.ibm.connector.CommunicationException: in the example above).

The sample then shows the message number and the method that throws this exception (HWSJ003E:com.ibm.connector.imstoc.IMSAdapter@7e91.connect() error:), along with the corresponding error message (Fail to connect to Host name[CSDMEC01], Port[9999]). Each exception message is identified by a message number (such as HWSJ003E). If an exception message is shown in brackets [] at the end of the error message (such as [java.net.SocketException: Connection timed out]), it shows the information of the base native exception that is caught and re-thrown according to the Common Connection Framework specification.

If a base native exception is caught and re-thrown by the connector, the printstack information reflects the printstack information of the base native exception. For specific exception details, see the Javadoc of the particular exception class.

Appendix D. Diagnosing problems related to sockets

If the IMS Connect environment appears to be suspended or the Java application or servlet fails to make a connection with an IMS Connect system that runs using a valid host name and port, you might have reached the maximum number of sockets. The number of sockets that can exist between a Java client (a Java application or servlet) and the host component, IMS Connect is controlled by two parameters that need to be synchronized. These parameters are the following:

Maxsoc

The maximum number of sockets for an IMS Connect port. **Maxsoc** applies to all of the ports associated with an instance of IMS Connect. **Maxsoc** is displayed in the **Maxsoc** field of the IMS Connect command **VIEWHWS**. The default value for **Maxsoc** is 50.

Important: The value displayed in the **Maxsoc** field reflects one socket dedicated to Listen State and the remainder available for connections. For example, if **Maxsoc** is 50, only 49 can be used by connections from Java clients (Java applications or servlets).

Example: If a particular host machine has a single instance of IMS Connect with 2 ports, and if **Maxsoc** is 50, each port can have 49 sockets (connections) from Java clients, yielding a maximum of 98 sockets into the IMS Connect instance.

When the **Maxsoc** limit is reached, IMS Connect enters a wait state, from which it returns at regular time intervals to check for available sockets. If the Java applications or servlets are not releasing their sockets, your environment can appear to be suspended. If this situation occurs, take one of the following actions:

- Consider increasing the maximum number of sockets.

  Related Reading: For information on increasing the maximum socket value, see the IMS Connect documentation at the IMS Web site:

  http://www.ibm.com/software/data/ims
- Make sockets available by cancelling one or more of the Java applications or servlets that are holding sockets to the particular IMS Connect.
- Avoid setting the value of the **disconnectedCommunication** property to false using the setDisconnectCommunication method of the com.ibm.ivj.eab.command.CommunicationCommand class. If the **disconnectedCommunication** property is false, the communication remains connected after execution.

Maximum connections

A property of the IMSConnectionSpec class that specifies the maximum number of connections (sockets) registered with the connection manager for the host name and port of the associated IMSConnectionSpec. In the case of the "real" connection manager (com.ibm.connector.connectionmanager.ConnectionManager), a pool of connections is kept for each host name and port combination. Therefore, **Maximum connections** determines the maximum number of connections for a pool.

After the **Maximum connections** number is reached, no new connections are created. The NoConnectionAvailableException is thrown.

© Copyright IBM Corp. 1998, 2000

Recommendation: When synchronizing **Maximum connections** and **Maxsoc**, it is **strongly recommended** that **Maximum connections** be less than or equal to **Maxsoc**.

Diagnosing problems related to connection pooling

If the connections used by your Java application or servlet are piled up, such that the connections are not reused by new transaction requests nor released during the end of transaction executions, you may not have connection pooling established appropriately in your environment. For information about connection pooling, see Connection Management (page 13).

Appendix E. Messages and exceptions

While you develop Java programs that use IMS Connector for Java, you might encounter situations in which your program throws exceptions. Some of these exceptions are thrown by IMS Connector for Java, while others are thrown by other class libraries (such as the IBM Common Connector Framework or the Java class libraries) that your program uses.

An example of an exception that is thrown by Java is:

```
java.lang.NoClassDefFoundError: com/ibm/connector/infrastructure/java/JavaRuntimeContext
```

A common cause for this exception is an incomplete class path. For VisualAge for Java, ensure that you compute the class path using the **Class path** tab of the **Properties** window for your application.

Related Reading: For information on exceptions that are thrown from other class libraries, see the Javadoc information for the specific class library.

When receiving a message from your application program, the term in *italics* represents one of the values that are inserted at run time.

hostname
The TCP/IP host name of the machine that is running IMS Connect.

innermethodname
The name of the method that originally throws this exception. This exception has been caught by IMS Connector for Java and is being re-thrown to another exception, according to the Common Connector Framework specification.

length The length of the data.

llvalue
The value of LL.

maxlength
The maximum valid length of the data.

messagecode
The message code of the exception.

methodname
The name of the method that is throwing this exception.

mode The type of interaction between IMS Connector for Java and the IMS Connect component on the host (as defined in the interactionspec).

portnumber
The port number that is assigned to IMS Connect.

propertyname
The name of the property.

propertyvalue
The value of the property.

reasoncode
The reason code that is returned by IMS Connect.

rectype
The type of the record.

© Copyright IBM Corp. 1998, 2000

returncode
The return code that is returned by IMS Connect or the sense code that is returned from IMS OTMA.

socketexception
The socket exception.

state The internal state of IMS Connector for Java.

HWSJ001E

**com.ibm.connector.imstoc.
IMSTOCResourceException:
HWSJ001E: methodname error.
IMS Connect returned error:
MESSAGECODE=[***messagecode***],
RETCODE=[***returncode***],
REASONCODE=[***reasoncode***].**

Explanation: An error code has been returned by IMS Connect on the host.

User Action: Related Reading: For diagnostic information on the return code (*returncode*) and reason code (*reasoncode*) values, see the *IMS Connect Messages and Codes* manual. Verify the host system using the information above.

HWSJ002E

**com.ibm.connector.imstoc.
IMSTOCResourceException:
HWSJ002E: methodname error.
IMS returned error:
MESSAGECODE=[***messagecode***],
RETCODE=[***returncode***].
IMS OTMA ERROR:
please see IMS OTMA Guide.**

Explanation: A NAK error message has been returned from IMS OTMA.

User Action: Related Reading: For diagnostic information on the sense code (*returncode*) value of the NAK message, see the *IMS OTMA Guide and Reference*. Verify the host system using the information above.

HWSJ003E

**com.ibm.connector.
CommunicationException:
HWSJ003E: methodname error.
Failed to connect to Host name
[***hostname***], Port [***portnumber***].
[***socketexception***].**

Explanation: *socketexception* is one of the following:

[java.net.UnknownHostException: *hostname***]**
Hostname is invalid. Check the spelling of the Host name, and use the fully qualified path for Host name. Correct the application, if necessary.

[java.net.SocketException: Connection refused]
One of the following occurred:

- Port number is invalid.

- The IMS Connect host component with the specified host name is down.
- TCP/IP restarted without canceling and restarting IMS Connect (HWS) on the host machine, or issuing **stopport** followed by **openport**.

Take the following actions:

- Verify that the port number for IMS Connect on the host with the specified host name.
- Start IMS Connect (HWS) on the host machine.
- Do one of the following:
  - Cancel and restart the IMS Connect component on the host machine.
  - Issue **stopport dddd**, followed by **openport dddd**, where **dddd** is the port number for IMS Connect.

[java.net.SocketException: Connection timed out]

One of the following occurred:

- The machine with the specified host name is inaccessible on the TCP/IP network.
- TCP/IP on the host system is down.

Take the following actions:

- Ensure that the host machine is accessible from the TCP/IP network. Verify by issuing the **ping** command to the host machine.
- Restart TCP/IP on the host and do one of the following:
  - Cancel and restart the IMS Connect component on the host machine
  - Issue **stopport dddd**, followed by **openport dddd**, where **dddd** is the port number for IMS Connect.

HWSJ004E **com.ibm.connector.CommunicationException: HWSJ004E: methodname error. Failed to close the connection. [*socketexception*]**

Explanation: The connection failed to close. The connection with the host has been reset, or the TCP/IP connection is down.

User Action: Refer to the *socketexception* for more details for the cause of failure. Restart IMS Connect and TCP/IP on the host, if appropriate.

HWSJ005E **com.ibm.connector.CommunicationException: HWSJ005E: methodname error. Failed to send the data. [*socketexception*]**

Explanation: Failed to write to a socket to send the transaction message to IMS Connect on the host. The connection with the host has been reset or the TCP/IP connection is down.

User Action: Refer the *socketexception* for more details of the failure. Restart IMS Connect and TCP/IP on the host, if appropriate.

HWSJ006E

**com.ibm.connector.
NotConnectedException:
HWSJ006E: methodname error.
The connection resource is not
connected.**

Explanation: An application programming error occurred while attempting to execute a transaction. The connection has not been established.

User Action: Correct the application program.

HWSJ007E

**com.ibm.connector.
AlreadyConnectedException:
HWSJ007E: methodname error.
The connection resource is already
connected.**

Explanation: An application programming error occurred while attempting to connect to IMS Connect on the host. The connection has already been established.

User Action: Correct the application program.

HWSJ008E

**java.lang.IllegalArgumentException:
HWSJ008E: methodname error.
The property [***propertyname***] exceeded
maximum length.
Length is [***length***],
maximum length is [***maxlength***].**

Explanation: An application programming error occurred. The property value is invalid (the length exceeded the maximum value).

User Action: Verify the property value. The length must be less than the maximum length value. Correct the application program.

HWSJ009E

**java.lang.IllegalArgumentException:
HWSJ009E: methodname error.
The [***propertyname***] property value
[***propertyvalue***] is invalid.**

Explanation: An application programming error occurred. The property value is invalid.

User Action: Verify the property value. Correct the application program.

Related Reading: For a list of supported values for property, see the VisualAge for Java help: **Help —> Reference —> IBM Tool APIs —> Connectors —> IMS Connect**.

HWSJ010E **java.lang.IllegalArgumentException: HWSJ010E: methodname error. The input record type is invalid. The type currently supported is [rectype].**

Explanation: An application programming error occurred. The type of input record bean is invalid.

User Action: Verify the type of input record bean. The record type of the input bean must be *rectype*. Correct the application program.

HWSJ011E **java.lang.IllegalArgumentException: HWSJ011E: methodname error. The output record type is invalid. The type currently supported is [*rectype*].**

Explanation: An application programming error occurred. The type of output record bean is invalid.

User Action: Verify the type of output record bean. The record type of the output bean must be *rectype*. Correct the application program.

HWSJ012E **com.ibm.connector. BadInvOrderException: HWSJ012E: methodname error. Protocol violation. The connection resource is in an inappropriate state [*state*] for the execution [*execution*].**

Explanation: An application programming error occurred during the ordered sequence of methods invocation of an interaction with IMS Connect on the host. Connector is in an inappropriate state for the requested execution. A possible cause might be that command.execute() is called before communication.connect().

User Action: Correct the application program. Correct the execution flow: communication.connect() —> communication.execute() —> communication.disconnect()

Related Reading: For the suggested execution flow, see the VisualAge for Java help: **Help —> Reference —> IBM Tool APIs —> Connectors —> IMS Connect**.

HWSJ013E **com.ibm.connector. BadInvOrderException: HWSJ013E: methodname error. Protocol violation. The Interaction Mode [*mode*] is not allowed for the current state [*state*].**

Explanation: An application programming error occurred in the ordered sequence of methods invocation of an interaction with IMS Connect on the host. The execution of the command with the specified interaction mode is not allowed in the state. A possible cause might be the execution of the second interaction of a command with the synchronization level of SYNC_LEVEL_CONFIRM (as specified in the IMSConnectionSpec bean) without previously executing an ACK or NACK command to reply to IMS Connect on the host for the first interaction.

For example, (1st interaction) command.execute() —> (2nd interaction) command.execute() ... An ACK or NACK command is an EAB command with interaction mode MODE_ACK or MODE_NACK.

User Action: Correct the application program. Correct the execution flow to the following: (1st interaction) command.execute() —> ackCommand.execute (to reply to IMS Connect on the host as it is waiting for a response because of the synchronization level of Confirm) —> (2nd interaction) command.execute()

Related Reading:

- For suggested execution flow, see the VisualAge for Java help: **Help —> Reference —> IBM Tool APIs —> Connectors —> IMS Connect**.

HWSJ014E **com.ibm.connector.ResourceException: HWSJ014E: methodname error. The connection resource is in an invalid state [*state*].**

Explanation: An application programming error occurred. The value of the **state** field is invalid.

User Action: Correct the application program.

Related Reading: For the valid values of **state**, see the VisualAge for Java help: **Help —> Reference —> IBM Tool APIs —> Connectors —> IMS Connect**.

HWSJ015E **java.lang.IllegalArgumentException: HWSJ015E: methodname error. This was an error in building the OTMA message. [*exception*]**

Explanation: The *exception* is one of the following:

-
```
com.ibm.imstoc.IMSException:
HWSJ501E: innermethodname error.
The LL value of the input message is incorrect.
```

  The **LL** value of the input message is incorrect. Ensure that the **LL** value of the input object that is provided by the application program matches the real length of the message data. Correct the application program.

-
```
com.ibm.imstoc.IMSException:
HWSJ502E: innermethodname error.
The input byte array is not pure DBCS.
There is an odd number of bytes.
```

The input byte array is not pure DBCS. An odd number of bytes exists. Ensure that this exception is being handled properly in your application program. See the exception for the specific cause of the error.

- For other exception types that are not listed above, see the *exception* of the specific cause of the error. An error building the OTMA message for the interaction request occurred. Ensure that the values provided by the application program are valid for building the interaction request. Correct the application program.

HWSJ016E **com.ibm.connector.ResourceException: HWSJ016E: methodname error. Two phase commit is not supported.**

Explanation: An application programming error occurred while attempting to use two-phase commit to coordinate the transaction. Currently, two-phase commit is not supported.

User Action: Use IMSConnection.commitOnePhase(). Correct the application program.

As suggested in the *IMS Connector for Java User's Guide*, the application implementation should invoke the Coordinator.commit() in order to commit a connection. If the application is using the original Coordinator class from CCF, the IMSConnection.commitOnePhase() method is automatically invoked. Therefore, this exception is not thrown. However, if the application uses the method (IMSConnection.commit()) directly to commit the connection, this exception is thrown.

HWSJ017E **com.ibm.connector. TransactionRolledBack: HWSJ017E: methodname error. An exception occurred during commit. Transaction was rolled back and the connection will be closed [*exception*].**

Explanation: An exception occurred while attempting to commit the transaction. This exception notifies the application program that the transaction has been rolled back and that the connection is closed.

Definition: A connection is marked as *dirty* when severe exceptions, such as communications or protocol violations, occur during the execution.

User Action: Ensure that this exception is handled properly by the application program. See the exception for the specific causes of the error.

HWSJ018E **HWSJ018E: methodname error. An exception occurred while attempting to close the dirty connection [*socketexception*].**

Explanation: An exception has occurred while attempting to close the dirty connection. This warning message is normally displayed after another exception is shown during the execution of the command.

Definition: A connection is marked as *dirty* when severe exceptions, such as communications or protocol violations, occur during the execution.

User Action: See the *socketexception* for the specific cause of the exception. Because this is a warning message, no specific action is needed. However, if the exception impacts the subsequent execution of the application program, ensure that the application can handle the situation properly.

HWSJ019E **com.ibm.connector.ResourceException: HWSJ019E: methodname error. Resource is registered with the coordinator.**

Explanation: An application programming error occurred. The connection resource is still registered with the Coordinator while preparing the connection to be reused.

User Action: Verify that the managed connection has been unregistered with the Coordinator by invoking commit()/rollback() before reusing the connection in the application program. Correct the application program.

HWSJ021E **com.ibm.connector.ResourceException: HWSJ021E: methodname error. Attempted to execute on a dirty connection.**

Explanation: An application programming error occurred while attempting to execute or commit on a dirty connection.

Definition: A connection is marked as *dirty* when severe exceptions, such as communications or protocol violations, occur during the execution.

If the application program attempts to commit on a dirty connection, this exception notifies the application program that the transaction has been rolled back.

User Action: Correct the application program. Ensure that this exception is handled properly in the application program. Do not perform any further execution using a dirty connection.

HWSJ022E **com.ibm.connector. TransactionRolledBack: HWSJ022E: methodname error. Attempted to commit on a dirty connection.**

Explanation: An application programming error occurred while attempting to commit on a dirty connection.

Definition: A connection is marked as *dirty* when severe exceptions, such as communications or protocol violations, occur during the execution.

User Action: Correct the application program. Ensure that this exception is handled properly by the application program. Do not perform any further execution using a dirty connection.

HWSJ024E

java.lang.IllegalArgumentException: HWSJ024E: methodname error. Invalid segment length (LL) of [*llvalue*] in input object.

Explanation: The segment length **LL** of the input object is invalid.

User Action: Verify that the segment length **LL** in the segment message matches the real length of the segment. Correct the application program.

HWSJ025E

java.lang.IllegalArgumentException: HWSJ025E: methodname error. Invalid segment length (LL) of [*llvalue*] in OTMA message.

Explanation: The segment length **LL** of the OTMA message is invalid while processing the output transaction message from IMS.

User Action: Verify that the transaction output message that is being sent back from the IMS transaction is correct. Verify that the segment length **LL** in the OTMA message that is received is valid and matches the real length of the segment. Correct the application program.

HWSJ026E

java.lang.IllegalArgumentException: HWSJ026E: methodname error. There was an error in processing the output message. [*exception*]

Explanation: An error occurred while processing the output message that is received from IMS. The output message that is sent from the IMS transaction might be invalid.

User Action: Verify that the output message that is sent from the IMS transaction is valid. See the exception for the specific cause of the error. Correct the application program.

HWSJ027E

java.lang.IllegalArgumentException: HWSJ027E: methodname error. An error occurred when completing the interaction request. [*exception*]

Explanation: An error occurred while completing the interaction request of the command. For example, the connection with IMS Connect on the host might be down while completing the interaction. The interaction request might or might not be completed successfully.

User Action: Ensure that this exception is handled properly by the application program. See the exception for the specific cause of the error.

HWSJ028E **com.ibm.connector.TransactionRolledBack: HWSJ028E: methodname error. An exception occurred during rollback. [***exception***]**

Explanation: An exception occurred while attempting to perform a roll back on the transaction. The connection is closed.

User Action: Ensure that this exception is handled properly by the application program. See the exception for the specific cause of the error.

HWSJ029E **com.ibm.connector.CommunicationException: HWSJ029E: methodname error. A communication error occurred while receiving the output message. [***exception***].**

Explanation: Failed to read the output message from the socket. One of the following may have occurred:

- The client on IMS Connect was closed. (For example, the IMS Connect **STOPCLNT** command was issued).
- The connection with the host has been reset or the TCP/IP connection is down.
- The output message is incomplete or corrupted.

Definition: A connection is marked as *dirty* when severe exceptions, such as communications or protocol violations, occur during the execution.

User Action: Refer to the *exception* for more details of the failure. The "dirty" connection will be closed and will not be reused. A new connection will be created for the next request.

Notices

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION ″AS IS″ WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright IBM Corp. 1998, 2000

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director
IBM Canada Ltd.
1150 Eglinton Avenue East
Toronto, Ontario M3C 1H7
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1997, 2000. All rights reserved.

Programming interface information

Programming interface information is intended to help you create application software using this program.

General-use programming interfaces allow the customer to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and service marks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

- AIX
- AS/400
- DB2
- CICS
- CICS/ESA
- IBM
- IMS
- Language Environment
- MQSeries
- Network Station
- OS/2
- OS/390
- OS/400
- RS/6000
- S/390
- VisualAge
- VTAM
- WebSphere

Lotus, Lotus Notes and Domino are trademarks or registered trademarks of Lotus Development Corporation in the United States, or other countries, or both.

Tivoli Enterprise Console and Tivoli Module Designer are trademarks of Tivoli Systems Inc. in the United States, or other countries, or both.

Encina and DCE Encina Lightweight Client are trademarks of Transarc Corporation in the United States, or other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

ActiveX, Microsoft, SourceSafe, Visual C++, Visual SourceSafe, Windows, Windows NT, Win32, Win32s and the Windows logo are trademarks or registered trademarks of Microsoft Corporation in the United States, or other countries, or both.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Intel and Pentium are trademarks of Intel Corporation in the United States, or other countries, or both.

Other company, product, and service names, which may be denoted by a double asterisk(**), may be trademarks or service marks of others.